# MyBatis概述1

使用JDBC持久化时，SQL语句被编码到代码中，耦合度太高，不易维护。而且开发中经常添加或修改SQL语句，只能到代码中去修改，更加不方便，因此出现了持久层框架来优化这个问题。

MyBatis 是支持定制化（半自动化）SQL、存储过程以及高级映射的半自动化持久层框架。避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。

MyBatis可以使用的XML或注解用于配置和原始映射，将接口和Java的POJO（Plain Old Java Objects）映射成数据库中的记录。

**补充**

Hibernate和JPA缺点：长难复杂SQL，对于Hibernate而言处理也不容易，内部自动生产的SQL，不容易做特殊优化，基于全映射的全自动框架，JavaBean存在大量字段时无法只映射部分字段，导致数据库性能下降。

对开发人员而言，核心SQL还是需要自己优化；SQL和编码分开，功能边界清晰，一个专注业务，一个专注数据。

## 使用步骤

1. 建立数据库及数据库表
2. 搭建mybatis开发环境，创建lib目录并导入mybatis和jdbc的jar包
3. 创建pojo目录，并创建JavaBean类
4. 创建mybatis-config.xml核心配置文件，一般创建在config源码目录中
5. 创建JavaBean类的配置文件，名称一般为JavaBean类名加上Mapper后加上文件后缀名xml
6. 创建dao及其实现类

## 四大部分

**mybatis-config.xml**

配置数据库链接信息

配置properties属性

配置mybatis全局设置

配置mybatis类的别名

配置mybatis的类型处理器

……

**XxxMapper.xml**

配置JavaBean对应数据库表的CRUD的sql语句

配置查询结果集的映射对象

配置一对一、一对多和多对多映射关系

**SqlSessionFactory**

创建SqlSession类

管理二级缓存

**SqlSession**

像数据库中的Connection对象

可以执行sql语句

管理jdbc事务和一级缓存

XxxMapper.xml需要在mybatis-config.xml中配置才能生效

SqlSessionFactory工厂类的创建需要读取mybatis-config.xml核心配置文件

SqlSession执行XxxMapper.xml中配置的sql语句，并且还可以由XxxMapper.xml文件生成一个Mapper类直接操作数据库

SqlSession类由SqlSessionFactroy类创建

## 入门案例

JavaBean

|  |
| --- |
| public class User {  private Integer id;  private String lastName;  private Integer sex; } |

mybatis-config.xml

该文件类型是xml文件，文件名是mybatis-config.xml，虽然文件名可以任意设置，但是约定俗成为该名称，便于阅读理解。

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8" ?> <!DOCTYPE configuration  PUBLIC "-//mybatis.org//DTD Config 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-config.dtd"> <configuration>  <environments default="development">  <environment id="development">  <transactionManager type="JDBC"/>  <dataSource type="POOLED">  <property name="driver" value="com.mysql.jdbc.Driver"/>  <property name="url" value="jdbc:mysql://localhost:3306/mybatis"/>  <property name="username" value="root"/>  <property name="password" value="root"/>  </dataSource>  </environment>  </environments>  <mappers>  <!-- 让mybatis的核心配置文件知道有哪些sql语句的配置文件 -->  <mapper resource="com/xmm/pojo/UserMapper.xml"/>  </mappers> </configuration> |

UserMapper.xml

该文件是xml类型的文件，因为JavaBean的类名为User，因此其配置的xml文件名为UserMapper.xml。两者对应便于阅读理解。

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8" ?> <!DOCTYPE mapper  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"  "http://mybatis.org/dtd/mybatis-3-mapper.dtd"> <!--namespace属性是名称空间，一般取值有两种： 第一种：javaBean的全类名  第二种：JavaBean对应的Dao接口（Mapper接口）的全类名 --> <mapper namespace="com.xmm.pojo.User">  <!--select 标签配置select查询语句,id 属性配置一个唯一的标识  resultType 属性配置select查询之后每一行记录将映射成为哪个Javabean对象(全类名)，#{id} 表示占位符 ? -->  <select id="selectUserById" resultType="com.xmm.pojo.User">  select id,last\_name lastName,sex from t\_user where id = #{id};  </select> </mapper> |

UserTest.java

|  |
| --- |
| @Test public void method() throws IOException {  // 从数据库mybatis中查询t\_user表中id为1的记录，读取配置文件，得到流对象  InputStream inputStream = Resources.*getResourceAsStream*("mybatis-config.xml");  //创建SqlSessionFactory对象实例，一般一个数据库只有一个SqlSessionFactory对象实例  SqlSessionFactory sessionFactory = new SqlSessionFactoryBuilder().build(inputStream);  //创建SqlSession对象，SqlSession相当于jdbc中的Connection，每次使用完都要及时关闭  SqlSession session = sessionFactory.openSession();  /\* selectOne是执行查询只返回一行记录的sql语句，  第一个参数是sql语句，sql语句是由名称空间和唯一的id值组成，第二个参数是sql语句对应的参数 \*/  User user = session.selectOne("com.xmm.pojo.User.selectUserById", 1);  System.*out*.println(user);  session.close(); } |

## 添加日志

导入log4j的jar包，在config源码目录下创建log4j.properties属性配置文件。rootLogger的值设置为DEBUG模式。

|  |
| --- |
| #将日志级别设置为DEBUG可以记录下其执行的SQL语句 log4j.rootLogger=DEBUG, stdout log4j.appender.stdout=org.apache.log4j.ConsoleAppender log4j.appender.stdout.layout=org.apache.log4j.PatternLayout log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n |

**log4j简单配置**

|  |  |
| --- | --- |
| **字符** | **含义** |
| %m | 输出代码中指定的消息 |
| %M | 输出打印该条日志的方法名 |
| %p | 输出优先级，即DEBUG，INFO，WARN，ERROR，FATAL |
| %r | 输出自应用启动到输出该log信息耗费的毫秒数 |
| %c | 输出所属的类目，通常就是所在类的全名 |
| %t | 输出产生该日志事件的线程名 |
| %n | 输出一个回车换行符，Windows平台为"rn”，Unix平台为"n” |
| %d | 输出日志时间点的日期或时间，默认格式为ISO8601，也可以在其后指定格式，比如：%d{yyyy-MM-dd HH:mm:ss,SSS}，输出类似：2002-10-18 22:10:28,921 |
| %l | 输出日志事件的发生位置，及在代码中的行数 |

# 不同CRUD方式

## 传统CRUD

传统方式的CRUD操作先要创建JavaBeanDao接口，然后创建JavaBeanDaoImpl实现类，再创建JavaBeanMapper配置文件，最后在service层调用实现类中的方法。

UserDao接口

|  |
| --- |
| public interface UserDao {  int saveUser(User user);  int deleteUserById(Integer id);  int updateUser(User user);  User queryUserById(Integer id);  List<User> queryUsers(); } |

UserDaoImpl实现类

|  |
| --- |
| public class UserDaoImpl implements UserDao {  private SqlSessionFactory sqlSessionFactory;  public UserDaoImpl(SqlSessionFactory sqlSessionFactory) {  this.sqlSessionFactory = sqlSessionFactory;  }  @Override  public int saveUser(User user) {  SqlSession session = sqlSessionFactory.openSession();  try{  //session对象的insert方法执行insert的sql语句  // 第一个参数表示要执行的sql语句,第二个参数表示要执行的sql语句的参数  // 返回值是sql语句影响的行数  int result = session.insert("com.hello.pojo.User.saveUser", user);  session.commit();  return result;  }finally {  session.close();  }  }  @Override  public int deleteUserById(Integer id) {  SqlSession session = sqlSessionFactory.openSession();  try{  int result = session.delete("com.hello.pojo.User.deleteUserById", id);  session.commit();  return result;  }finally {  session.close();  }  }  @Override  public int updateUser(User user) {  SqlSession session = sqlSessionFactory.openSession();  try{  int result = session.update("com.hello.pojo.User.updateUser", user);  session.commit();  return result;  }finally {  session.close();  }  }  @Override  public User queryUserById(Integer id) {  SqlSession session = sqlSessionFactory.openSession();  try{  //selectOne方法查询只返回一行（一个JavaBean对象）的sql语句  return session.selectOne("com.hello.pojo.User.queryUserById",id);  }finally {  session.close();  }  }  @Override  public List<User> queryUsers() {  SqlSession session = sqlSessionFactory.openSession();  try{  //selectList查询返回多行（多个JavaBean对象）的sql语句  return session.selectList("com.hello.pojo.User.queryUsers");  }finally {  session.close();  }  } } |

UserMapper.xml配置文件

|  |
| --- |
| <mapper namespace="com.hello.pojo.User">  <!-- insert标签配置insert语句  id是给sql配置的唯一标识，一般和调用该sql语句的方法名相同  parameterType设置方法的参数类型，一般是JavaBean的全类名或基本数据类型-->  <insert id="saveUser" parameterType="com.hello.pojo.User">  insert into t\_user(`last\_name`,`sex`) values(#{lastName},#{sex});  </insert>  <!-- delete标签配置delete语句 -->  <delete id="deleteUserById" >  delete from t\_user where id = #{id}  </delete>  <!-- update标签配置update语句 -->  <update id="updateUser" parameterType="com.hello.pojo.User">  update t\_user set last\_name = #{lastName},sex = #{sex} where id=#{id}  </update>  <!-- select标签配置select语句  resultType是配置select查询完之后结果集中每一行记录转换的JavaBean对象的全类名-->  <select id="queryUserById" resultType="com.hello.pojo.User">  select id,last\_name lastName,sex from t\_user where id = #{id}  </select>  <select id="queryUsers" resultType="com.hello.pojo.User">  select id,last\_name lastName,sex from t\_user  </select> </mapper> |

mybatis-config.xml配置文件

|  |
| --- |
| <mappers>  <mapper resource="com/hello/pojo/UserMapper.xml"/> </mappers> |

## Mapper接口方式CRUD（推荐）

使用mapper接口方式进行crud时不用创建mapper接口的实现类，可以直接调用对应mapper.xml的sql语句来实现crud操作,从而达到简化代码，提高编程效率的作用。

**命名规范**

对于mapper接口编程，有一套约定俗成的命名规范，针对JavaBean的类名来确定接口名和配置文件名

|  |  |  |
| --- | --- | --- |
| **JavaBean** | **接口名** | **SQL语句配置文件名** |
| User | UserMapper.java | UserMapper.xml |
| Book | BookMapper.java | BookMapper.xml |

**开发规范**

mapper.xml配置文件中名称空间必须是Mapper接口的全类名

接口中方法名必须和Mapper.xml配置文件中的id值一致

接口的参数类型必须要和Mapper.xml中配置的参数类型一致

接口的方法的返回值类型（JavaBean）一定要和mapper.xml中配置的ResultType类型一致

Mapper接口

|  |
| --- |
| public interface UserMapper {  public int saveUser(User user);  public int deleteUserById(Integer id);  public int updateUser(User user);  public User queryUserById(Integer id);  public List<User> queryUsers(); } |

Mapper.xml配置文件

|  |
| --- |
| <mapper namespace="com.hello.mapper.UserMapper">  <insert id="saveUser" parameterType="com.hello.pojo.User">  insert into t\_user(`last\_name`,`sex`) values(#{lastName},#{sex});  </insert>  <delete id="deleteUserById" >  delete from t\_user where id = #{id}  </delete>  <update id="updateUser" parameterType="com.hello.pojo.User">  update t\_user set last\_name = #{lastName},sex = #{sex} where id=#{id}  </update>  <select id="queryUserById" resultType="com.hello.pojo.User">  select id,last\_name lastName,sex from t\_user where id = #{id}  </select>  <select id="queryUsers" resultType="com.hello.pojo.User">  select id,last\_name lastName,sex from t\_user  </select> </mapper> |

mybatis-config.xml配置文件

|  |
| --- |
| <mappers>  <mapper resource="com/hello/mapper/UserMapper.xml"/> </mappers> |

|  |
| --- |
| public class UserMapperTest {  static SqlSessionFactory *sqlSessionFactory*;  @BeforeClass  public static void setUpBeforeClass() throws Exception {  *sqlSessionFactory* = new SqlSessionFactoryBuilder().build(Resources.*getResourceAsStream*("mybatis-config.xml"));  }  @Test  public void testSaveUser() {  SqlSession session = *sqlSessionFactory*.openSession();  try {  // 获取UserMapper接口的一个实现类（Mybatis底层自动实现jdk动态代理）  UserMapper mapper = session.getMapper(UserMapper.class);  mapper.saveUser(new User(null, "Tom", 0));  //对于增删改操作而言，执行完sql语句需要提交事务，使执行的操作生效  //而对于查询操作而言则不需要提交事务  session.commit();  } finally {  session.close();  }  }  @Test  public void testDeleteUserById() {  SqlSession session = *sqlSessionFactory*.openSession();  try {  UserMapper mapper = session.getMapper(UserMapper.class);  mapper.deleteUserById(7);  session.commit();  } finally {  session.close();  }  }  @Test  public void testUpdateUser() {  SqlSession session = *sqlSessionFactory*.openSession();  try {  UserMapper mapper = session.getMapper(UserMapper.class);  mapper.updateUser(new User(10, "abcde", 1));  session.commit();  } finally {  session.close();  }  }  @Test  public void testQueryUserById() {  SqlSession session = *sqlSessionFactory*.openSession();  try {  UserMapper mapper = session.getMapper(UserMapper.class);  System.*out*.println(mapper.queryUserById(1));  } finally {  session.close();  }  }  @Test  public void testQueryUsers() {  SqlSession session = *sqlSessionFactory*.openSession();  try {  UserMapper mapper = session.getMapper(UserMapper.class);  List<User> queryUsers = mapper.queryUsers();  for (User user : queryUsers) {  System.*out*.println(user);  }  } finally {  session.close();  }  } } |

## 注解方式CRUD

使用注解的方式进行CRUD在实际开发中很少使用，因为这种方式下如果要修改sql语句还是要在Java源代码中修改，耦合度太高，所以一般都是使用xml配置文件的方式进行CRUD。

|  |
| --- |
| public interface UserMapper {  @Insert("insert into t\_user(`last\_name`,`sex`) values(#{lastName},#{sex})")  public int saveUser(User user);  @Delete("delete from t\_user where id = #{id}")  public int deleteUserById(Integer id);  @Update("update t\_user set `last\_name`=#{lastName},`sex`=#{sex} where id = #{id}")  public int updateUser(User user);  @Select("select `id`,`last\_name`,`sex` from t\_user where id = #{id}")  public User queryUserById(Integer id);  @Select("select `id`,`last\_name`,`sex` from t\_user")  public List<User> queryUsers(); } |

通过注解方式进行CRUD时调用方法和使用Mapper.xml方式相同，都是通过SqlSession的getMapper方法获取UserMapper接口的实现类，然后通过该实现类来调用UserMapper接口中的方法。在执行增删改的SQL操作时都要调用commit方法进行事务的提交。

# MyBatis核心配置

mybatis核心配置文件mybatis-config.xml中有许多可以用于配置属性的标签，且这些标签在xml文件中的配置顺序是固定的，其顺序如下：

properties->settings->typeAliases->typeHandlers->objectFactory->objectWrapperFactory->reflectorFactory->plugins->environments->databaseIdProvider->mappers

## properties

properties配置了多组键值对供mybatis程序使用，resource用来读取指定配置文件的键值对信息。当resource属性中指定的属性配置文件中有和property配置相同名称的键的键值对时，以resource引入的文件为主，resource中引入的jdbc.properties属性会覆盖掉property标签中定义的属性。

|  |
| --- |
| <properties resource="jdbc.properties">  <property name="username" value="root"/>  <property name="password" value="root"/> </properties> |

通过properties标签设置配置信息后可以将environments标签中的property标签中value属性值修改为EL表达式。

|  |
| --- |
| <environments default="development">  <environment id="development">  <transactionManager type="JDBC"/>  <dataSource type="POOLED">  <property name="driver" value="${driverClass}"/>  <property name="url" value="${url}"/>  <property name="username" value="${username}"/>  <property name="password" value="${password}"/>  </dataSource>  </environment> </environments> |

## settings

settings配置是mybatis中极为重要的设置，其设置会改变mybatis的运行时行为。

|  |
| --- |
| <settings>  <setting name="mapUnderscoreToCamelCase" value="true"/> </settings> |

settings设置选项

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| cacheEnabled | | | | | | | true | false | | | | true | | |
| 该配置影响的所有映射器中配置的缓存的全局开关。 | | | | | | | | | | | | | |
| lazyLoadingEnabled | | | | | | | true | false | | | | false | | |
| 延迟加载的全局开关。当开启时，所有关联对象都会延迟加载。 特定关联关系中可通过设置fetchType属性来覆盖该项的开关状态。 | | | | | | | | | | | | | |
| aggressiveLazyLoading | | | | | | | true | false | | | | true | | |
| 当启用时，对任意延迟属性的调用会使带有延迟加载属性的对象完整加载；反之，每种属性将会按需加载。 | | | | | | | | | | | | | |
| multipleResultSetsEnabled | | | | | | | true | false | | | | true | | |
| 是否允许单一语句返回多结果集（需要兼容驱动）。 | | | | | | | | | | | | | |
| useColumnLabel | | | | | | | true | false | | | | true | | |
| 使用列标签代替列名。不同的驱动在这方面会有不同的表现， 具体可参考相关驱动文档或通过测试这两种不同的模式来观察所用驱动的结果。 | | | | | | | | | | | | | |
| useGeneratedKeys | | | | | | | true | false | | | | False | | |
| 允许 JDBC 支持自动生成主键，需要驱动兼容。 如果设置为 true 则这个设置强制使用自动生成主键，尽管一些驱动不能兼容但仍可正常工作（比如 Derby）。 | | | | | | | | | | | | | |
| autoMappingBehavior | | | | | | | NONE, PARTIAL, FULL | | | | PARTIAL | | |
| 指定 MyBatis 应如何自动映射列到字段或属性。 NONE 表示取消自动映射；PARTIAL 只会自动映射没有定义嵌套结果集映射的结果集。 FULL 会自动映射任意复杂的结果集（无论是否嵌套）。 | | | | | | | | | | | | | |
| autoMappingUnknownColumnBehavior | | | | | | | NONE, WARNING, FAILING | | | | NONE | | |
| Specify the behavior when detects an unknown column (or unknown property type) of automatic mapping target.  NONE: Do nothing  WARNING: Output warning log (The log level of'org.apache.ibatis.session.AutoMappingUnknownColumnBehavior'must be set to WARN)  FAILING: Fail mapping (Throw SqlSessionException) | | | | | | | | | | | | | |
| defaultExecutorType | | | | | | | SIMPLE REUSE BATCH | | | | SIMPLE | | |
| 配置默认的执行器。SIMPLE 就是普通的执行器；REUSE 执行器会重用预处理语句（prepared statements）； BATCH 执行器将重用语句并执行批量更新。 | | | | | | | | | | | | | |
| defaultStatementTimeout | | | | | | | Any positive integer | | | | Not Set (null) | | |
| 设置超时时间，它决定驱动等待数据库响应的秒数。 | | | | | | | | | | | | | |
| defaultFetchSize | | | | | | | Any positive integer | | | | Not Set (null) | | |
| Sets the driver a hint as to control fetching size for return results. This parameter value can be override by a query setting. | | | | | | | | | | | | | |
| safeRowBoundsEnabled | | | | | | | true | false | | | | False | | |
| 允许在嵌套语句中使用分页（RowBounds）。 If allow, set the false. | | | | | | | | | | | | | |
| safeResultHandlerEnabled | | | | | | | true | false | | | | True | | |
| 允许在嵌套语句中使用分页（ResultHandler）。 If allow, set the false. | | | | | | | | | | | | | |
| mapUnderscoreToCamelCase | | | | | | | true | false | | | | False | | |
| 是否开启自动驼峰命名规则（camel case）映射，即从经典数据库列名 A\_COLUMN 到经典 Java 属性名 aColumn 的类似映射。 | | | | | | | | | | | | | |
| localCacheScope | | | | | | | SESSION | STATEMENT | | | | SESSION | | |
| MyBatis 利用本地缓存机制（Local Cache）防止循环引用（circular references）和加速重复嵌套查询。 默认值为 SESSION，这种情况下会缓存一个会话中执行的所有查询。 若设置值为 STATEMENT，本地会话仅用在语句执行上，对相同 SqlSession 的不同调用将不会共享数据。 | | | | | | | | | | | | | |
| jdbcTypeForNull | | | | JdbcType enumeration. Most common are: NULL, VARCHAR and OTHER | | | | | | | | OTHER | |
| 当没有为参数提供特定的 JDBC 类型时，为空值指定 JDBC 类型。 某些驱动需要指定列的 JDBC 类型，多数情况直接用一般类型即可，比如 NULL、VARCHAR 或 OTHER。 | | | | | | | | | | | | | |
| lazyLoadTriggerMethods | | | | | A method name list separated by commas | | | | | equals,clone,hashCode,toString | | | |
| 指定哪个对象的方法触发一次延迟加载。 | | | | | | | | | | | | | |
| defaultScriptingLanguage | | | | | | A type alias or fully qualified class name. | | | org.apache.ibatis.  scripting.xmltags.XMLDynamicLanguageDriver | | | | |
| 指定动态 SQL 生成的默认语言。 | | | | | | | | | | | | | |
| callSettersOnNulls | | | | | | | true | false | | | | false | | |
| 指定当结果集中值为null的时候是否调用映射对象的setter（map对象时为put）方法，这对于有Map.keySet() 依赖或 null 值初始化的时候是有用的。注意基本类型（int、boolean等）是不能设置成 null 的。 | | | | | | | | | | | | | |
| logPrefix | | | | | | | Any String | | | | Not set | | |
| 指定 MyBatis 增加到日志名称的前缀。 | | | | | | | | | | | | | |
| logImpl | | SLF4J|LOG4J|LOG4J2|JDK\_LOGGING|COMMONS\_LOGGING|STDOUT\_LOGGING|NO\_LOGGING | | | | | | | | | | Not set | |
| 指定 MyBatis 所用日志的具体实现，未指定时将自动查找。 | | | | | | | | | | | | | |
| proxyFactory | | | CGLIB | JAVASSIST | | | | | JAVASSIST (MyBatis 3.3 or above) | | | | | |
| 指定 Mybatis 创建具有延迟加载能力的对象所用到的代理工具。 | | | | | | | | | | | | | |
| vfsImpl | Fully qualified class names of custom VFS implementation separated by commas. | | | | | | | | | | | | Not set |
| Specifies VFS implementations | | | | | | | | | | | | | |
| useActualParamName | | | | | | | true | false | | | | true | | |
| Allow referencing statement parameters by their actual names declared in the method signature. To use this feature, your project must be compiled in Java 8 with -parameters option. (Since: 3.4.1) | | | | | | | | | | | | | |

## typeAliases

**自定义别名**

typeAliases标签可以配置一个别名，也可以配置多组别名，一般设置的别名命名规范是JavaBean的类名的首字母小写。类型别名是为Java类型设置一个短的名字，只和XML配置有关，存在的意义仅在于减少类名的冗余。

如果需要配置别名的JavaBean类过多，可以通过指定一个包名，让mybatis自动查找所有类并自动配置别名，自动配置的别名就是JavaBean的类名。

|  |
| --- |
| <typeAliases>  <typeAlias type="com.xmm.pojo.User" alias="user"/>  <package name="com.xmm.pojo"/> </typeAliases> |

当多个包下有相同名称的JavaBean时，可以使用注解@Alias来自定义别名。

|  |
| --- |
| @Alias("tUser") public class User {  private Integer id;  private String lastName;  private Integer sex; } |

**预设别名**

mybatis中为许多常见的Java类型内建了相应的类型别名，且都是大小写不敏感的，需要注意的是由基本类型名称重复导致的特殊处理。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **别名** | **映射的类型** | **别名** | **映射的类型** | **别名** | **映射的类型** |
| \_byte | byte | \_long | long | \_short | short |
| \_int|\_integer | int | \_double | double | \_float | float |
| \_boolean | boolean | string | String | byte | Byte |
| long | Long | short | Short | int|integer | Integer |
| double | Double | float | Float | boolean | Boolean |
| date | Date | decimal|bigdecimal | BigDecimal | object | Object |
| map | Map | hashmap | HashMap | list | List |
| arraylist | ArrayList | collection | Collection | iterator | Iterator |

## typeHandlers

无论是 MyBatis 在预处理语句（PreparedStatement）中设置一个参数时，还是从结果集中取出一个值时， 都会用类型处理器将获取的值以合适的方式转换成 Java 类型。下表描述了一些默认的类型处理器。

|  |  |  |
| --- | --- | --- |
| **类型处理器** | **Java 类型** | **JDBC 类型** |
| BooleanTypeHandler | java.lang.Boolean,boolean | 数据库兼容的 BOOLEAN |
| ByteTypeHandler | java.lang.Byte, byte | 数据库兼容的 NUMERIC 或 BYTE |
| ShortTypeHandler | java.lang.Short, short | 数据库兼容的 NUMERIC 或 SHORT INTEGER |
| IntegerTypeHandler | java.lang.Integer, int | 数据库兼容的 NUMERIC 或 INTEGER |
| LongTypeHandler | java.lang.Long, long | 数据库兼容的 NUMERIC 或 LONG INTEGER |
| FloatTypeHandler | java.lang.Float, float | 数据库兼容的 NUMERIC 或 FLOAT |
| DoubleTypeHandler | java.lang.Double, double | 数据库兼容的 NUMERIC 或 DOUBLE |
| BigDecimalTypeHandler | java.math.BigDecimal | 数据库兼容的 NUMERIC 或 DECIMAL |
| StringTypeHandler | java.lang.String | CHAR, VARCHAR |
| ClobReaderTypeHandler | java.io.Reader | - |
| ClobTypeHandler | java.lang.String | CLOB, LONGVARCHAR |
| NStringTypeHandler | java.lang.String | NVARCHAR, NCHAR |
| NClobTypeHandler | java.lang.String | NCLOB |
| BlobInputStreamTypeHandler | java.io.InputStream | - |
| ByteArrayTypeHandler | byte[] | 数据库兼容的字节流类型 |
| BlobTypeHandler | byte[] | BLOB, LONGVARBINARY |
| DateTypeHandler | java.util.Date | TIMESTAMP |
| DateOnlyTypeHandler | java.util.Date | DATE |
| TimeOnlyTypeHandler | java.util.Date | TIME |
| SqlTimestampTypeHandler | java.sql.Timestamp | TIMESTAMP |
| SqlDateTypeHandler | java.sql.Date | DATE |
| SqlTimeTypeHandler | java.sql.Time | TIME |
| ObjectTypeHandler | Any | OTHER 或未指定类型 |
| EnumTypeHandler | Enumeration Type | VARCHAR-任何兼容的字符串类型，存储枚举的名称（而不是索引） |
| EnumOrdinalTypeHandler | Enumeration Type | 任何兼容的 NUMERIC 或 DOUBLE 类型，存储枚举的索引（而不是名称）。 |

## objectFactory

MyBatis 每次创建结果对象的新实例时，它都会使用一个对象工厂（ObjectFactory）实例来完成。 默认的对象工厂需要做的仅仅是实例化目标类，要么通过默认构造方法，要么在参数映射存在的时候通过参数构造方法来实例化。 如果想覆盖对象工厂的默认行为，则可以通过创建自己的对象工厂来实现。

|  |
| --- |
| public class MyFactory extends DefaultObjectFactory {  public Object create(Class type) {  return super.create(type);  }  public Object create(Class type, List constructorArgTypes, List constructorArgs) {  return super.create(type, constructorArgTypes, constructorArgs);  }  public void setProperties(Properties properties) {  super.setProperties(properties);  }  public <T> boolean isCollection(Class<T> type) {  return Collection.class.isAssignableFrom(type);  } } |

|  |
| --- |
| <objectFactory type="com.xmm.factory.MyFactory">  <property name="someProperty" value="100"/> </objectFactory> |

ObjectFactory 接口包含两个创建用的方法，一个是处理空参构造器，另外一个是处理有参构造器。 setProperties 方法可以用来配置 ObjectFactory，初始化 ObjectFactory 实例后，objectFactory 元素体中定义的属性会被传递给 setProperties 方法。

## plugins

MyBatis 允许在已映射语句执行过程中的某一点进行拦截调用。默认情况下，MyBatis 允许使用插件来拦截的方法调用包括：

Executor负责update,query,flushStatements,commit,rollback,getTransaction,close,isClosed

ParameterHandler负责getParameterObject,setParameters

ResultSetHandler负责handleResultSets,handleOutputParameters

StatementHandler负责prepare,parameterize,batch,update,query

通过 MyBatis 提供的强大机制，使用插件是非常简单的，只需实现 Interceptor 接口，并指定了想要拦截的方法签名即可。

InterceptorChain过滤器链，管理和调用全部的过滤器执行

Interceptor过滤器，负责拦截Executor、ParameterHandler、ResultsetHandler、StatementHandler

|  |
| --- |
| @Intercepts({@Signature(type= Executor.class, method = "update", args = {MappedStatement.class,Object.class})}) public class ExamplePlugin implements Interceptor {  public Object intercept(Invocation invocation) throws Throwable {  return invocation.proceed();  }  public Object plugin(Object target) {  return Plugin.*wrap*(target, this);  }  public void setProperties(Properties properties) {  } } |

|  |
| --- |
| <plugins>  <plugin interceptor="com.xmm.plugins.ExamplePlugin">  <property name="someProperty" value="100"/>  </plugin> </plugins> |

上面的插件将会拦截在 Executor 实例中所有的“update”方法调用，这里的Executor是负责执行低层映射语句的内部对象。

除了用插件来修改MyBatis核心行为之外，还可以通过完全覆盖配置类来达到目的。只需继承后覆盖其中的每个方法，再把它传递到sqlSessionFactoryBuilder.build(myConfig)方法即可。再次重申，这可能会严重影响 MyBatis 的行为，务请慎之又慎。

## environments

enviroments标签可以配置多个数据库环境，default属性用来设置当有多个数据库环境时要使用的具体的数据库环境。一般开发中会定义多个环境信息，可以在开发和运行时分别使用不同的数据库。

enviroment标签可以配置一个数据库环境。其属性id的值可以任意取值，但是最好要见名知意。

transactionManager标签可以配置事务管理器，其属性type的值有两种

JDBC时表示有事务的提交和回滚，依赖于从数据源得到的连接来管理事务范围。

MANAGED时表示没有事务的提交和回滚，将事务交给容器处理。它从来不提交或回滚一个连接，而是让容器来管理事务的整个生命周期（比如 JEE 应用服务器的上下文）。 默认情况下它会关闭连接，然而一些容器并不希望这样，因此需要将 closeConnection 属性设置为 false 来阻止它默认的关闭行为。例如:

|  |
| --- |
| <transactionManager type="MANAGED">  <property name="closeConnection" value="false"/> </transactionManager> |

dataSource标签可以配置数据链接的使用方式，其属性type的值有三种

UNPOOLED表示每访问一个数据库，建立一个链接对象

POOLED表示使用数据库连接池技术，在开发中一般都会设置为POOLED来提高效率和用户体验

JNDI现已基本不用

自定义实现DataSourceFactory接口，定义数据源实现

|  |
| --- |
| <environments default="development">  <environment id="development">  <transactionManager type="JDBC"/>  <dataSource type="POOLED">  <!-- 数据库的四个连接属性 -->  <property name="driver" value="com.mysql.jdbc.Driver"/>  <property name="url" value="jdbc:mysql://localhost:3306/mybatis"/>  <property name="username" value="root"/>  <property name="password" value="root"/>  </dataSource>  </environment> </environments> |

## databaseIdProvider

mybatis可以根据不同的数据库厂商执行不同的语句，这种多厂商的支持是基于映射语句中的databaseId属性，mybatis会加载不带 databaseId 属性和带有匹配当前数据库 databaseId 属性的所有语句。

mybatis提供了一个类VendorDatabaseIdProvider，中的getDatabaseId() 方法用于获取数据库的标识。

property 标签name属性是获取数据库ID标识，value属性是给mybatis定义的一个简短的标识。

|  |
| --- |
| <databaseIdProvider type="DB\_VENDOR">  <property name="SQL Server" value="sqlserver"/>  <property name="MySQL" value="mysql"/>  <property name="DB2" value="db2"/>  <property name="Oracle" value="oracle"/> </databaseIdProvider> |

databaseId属性设置

|  |
| --- |
| <!-- databaseId表示当前sql语句只有SQL Server数据库才会使用 --> <select id="queryUsers" resultType="com.xmm.pojo.User" databaseId="sqlserver">  select id,last\_name lastName,sex from t\_user </select> <!-- databaseId表示当前sql语句只有MySQL数据库才会使用 --> <select id="queryUsers" resultType="com.xmm.pojo.User" databaseId="mysql">  select id,last\_name lastName,sex from t\_user </select> <!-- databaseId表示当前sql语句只有Oracle数据库才会使用 --> <select id="queryUsers" resultType="com.xmm.pojo.User" databaseId="oracle">  select id,last\_name lastName,sex from t\_user </select> |

## mapper

mapper标签用来通知mybatis都有哪些sql语句的配置文件，共有四种方式，如下所示：

1、从类路径下加载指定的mapper.xml

2、使用全类名加载mapper.xml，此时要求mapper接口和mapper.xml同包名

3、使用包名加载mapper.xml，指定一个包名，然后mybatis自动搜索全部的mapper.xml，要求接口和mapper.xml同包名

|  |
| --- |
| <mappers>  <mapper resource="com/xmm/mapper/UserMapper.xml"/>  <mapper class="com.xmm.mapper.UserMapper"/>  <package name="com.xmm.mapper"/> </mappers> |

还有一种通过file协议指定绝对路径来加载mapper.xml的方式不推荐使用，因为实际web开发中一旦将程序部署到服务器上，file协议就失去作用，导致无法加载mapper.xml。

# SQL语句参数传递

## 普通数据类型

当SQL语句中传入的参数是String、包装类等普通数据类型时，#{}中填写的名称如下：

**单个参数**

当传入的参数类型是一个普通类型时，只需要在占位符#{}中任意填写即可。但是为了见名知义，推荐以参数名填写。

**UserMapper.java**

|  |
| --- |
| public interface UserMapper {  User queryUserById(Integer id); } |

**UserMapper.xml**

|  |
| --- |
| <!-- #{}可以填写任意内容(不能不写)，但是建议填写参数名，增强可读性 --> <select id="queryUserById" resultType="com.hello.pojo.User">  select id,last\_name lastName,sex from t\_user where id = #{id}; </select> |

**多个参数**

当传入的参数是多个普通类型参数时，#{}中填写的内容有两种方案，如下：

1. 第一个参数写为0，第二个参数写为1，以此类推，第n个参数写为n-1
2. 第一个参数写为param1，第二个参数写为param2，以此类推，第n个参数写为paramN

**UserMapper.java**

|  |
| --- |
| public interface UserMapper {  List<User> queryUsersByNameOrSex(String name, Integer sex); } |

**UserMapper.xml**

|  |
| --- |
| <select id="queryUsersByNameOrSex" resultType="com.hello.pojo.User">  select id,last\_name lastName,sex from t\_user where last\_name like #{param1} or sex = #{param2} </select> |

上述两种方式不能见名知义，程序可读性较差，因此可以在mapper接口中方法的参数上添加注解@Param来自定义sql语句中#{}填写的内容。

**UserMapper.java**

|  |
| --- |
| public interface UserMapper {  User queryUserById(Integer id);  List<User> queryUsersByNameOrSex(@Param("name") String name, @Param("sex") Integer sex); } |

**UserMapper.xml**

|  |
| --- |
| <select id="queryUsersByNameOrSex" resultType="com.hello.pojo.User">  select id,last\_name lastName,sex from t\_user where last\_name like #{name} or sex = #{sex} </select> |

## Map对象

当传入的参数是map类型的对象时，#{}中的内容必须要和map的key值保持一致。

**UserMapper.java**

|  |
| --- |
| public interface UserMapper {  public List<User> queryUsersByMap(Map<String,Object> param); } |

**UserMapper.xml**

|  |
| --- |
| <!-- Map<String,Object> map = new HashMap<String, Object>();  map.put("name", "xmm");  map.put("sex", 0); --> <select id="queryUsersByMap" resultType="com.param.pojo.User">  select id,last\_name lastName,sex from t\_user where last\_name like #{name} or sex = #{sex} </select> |

## JavaBean对象

**单个参数**

当传入的参数是一个JavaBean对象时，#{}中的内容是JavaBean的属性名

**UserMapper.java**

|  |
| --- |
| public interface UserMapper {  public List<User> queryUsersByUser(User user); } |

**UserMapper.xml**

|  |
| --- |
| <select id="queryUsersByUser" resultType="com.param.pojo.User">  select id,last\_name lastName,sex from t\_user where last\_name like #{lastName} or sex = #{sex} </select> |

**多个参数**

当传入的参数是多个JavaBean对象时，param1表示第一个JavaBean对象，param2表示第二个JavaBean对象，以此类推。此时#{}中填写的内容如果是第一个JavaBean的属性attr1则为param1.attr1，第二个JavaBean的属性attr2则为param2.attr2。

也可以使用@Param注解命名参数，给每一个JavaBean对象起一个别名，然后使用#{别名.属性名}的方式作为参数使用。

**UserMapper.java**

|  |
| --- |
| public interface UserMapper {  public List<User> queryUsersByUsers(User user1,User user2); } |

**UserMapper.xml**

|  |
| --- |
| <select id="queryUsersByUsers" resultType="com.param.pojo.User">  select id,last\_name lastName,sex from t\_user where last\_name like #{param1.lastName} or sex = #{param2.sex} </select> |

## 模糊查询

一般来说根据用户名查询用户对象用的都是模糊查询，即select \* from t\_user where user\_name like '%输入名称%'，此时就要考虑到sql语句中的字符串拼接问题，一般使用sql语句中的concat函数来进行字符串拼接。

**UserMapper.java**

|  |
| --- |
| public interface UserMapper {  public List<User> queryUsersByName(String name); } |

**UserMapper.xml**

|  |
| --- |
| <select id="queryUsersByName" resultType="com.param.pojo.User">  select id,last\_name lastName,sex from t\_user where last\_name like *concat*('%',#{name},'%') </select> |

## #{}和${}的区别

#{}是mapper配置文件中sql语句中的占位符，作用和jdbc中的sql语句中的？占位符相同。它原样输出变量的值，然后以字符串拼接的功能进行操作。

${}是将参数的值value直接输入，或者是@Param命名参数后的参数名称，然后和原先的sql语句做字符串拼接操作，示例如下：

where last\_name like '%${value}%'，把参数List<User> users = mapper.queryUsersByName("aaa"); 输入之后是:where last\_name like '%aaa%'

在输出参数时，并不推荐使用${}来输出，可能会导致sql注入的问题存在。

在实际开发中，更多的是使用sql中的concat函数来做字符串拼接操作，而不是使用${}和#{}来做字符串拼接。使用concat函数的好处在于传入参数时不用再手动添加%，而且避免了sql注入的问题。

# MyBatis自定义结果集

自定义结果集可以给复杂的对象使用，即对象内又嵌套一个对象或者一个集合。

<resultMap>标签可以将查询回来的结果集封装称为复杂的JavaBean对象，其id属性用来设置一个唯一的标识，type属性设置每一行封装成的JavaBean对象类型（全类名）。

简单的JavaBean对象：只有基本类型的属性，一般使用resultType进行封装。

复杂的JavaBean对象：含有自定义对象类型或集合类型的属性，一般使用resultMap进行封装。

<id>标签用来映射主键的列和属性，其column属性标识映射的表中的列，property属性映射的JavaBean中的属性。

<result>标签用来映射非主键的列和属性，其column和property属性作用于id相同。

<association>标签可以将JavaBean中的子对象使用一个查询来返回其值，其property属性设置要封装的子对象，select通过全类名.方法名的方式设置要调用的查询方法，column将某个列的值传递给select查询作为参数，javaType设置当前查询结果返回的JavaBean类型。

<collection>标签可以用来映射封装一个集合对象，也可以调用一个查询sql语句，得到需要的集合信息。其property属性设置封装的目标集合；ofType设置集合中每个元素的具体类型；select设置调用的sql语句查询，其方式同association标签的select属性方式相同；column设置作为参数传递的结果集中的某一列。

## 一对一

|  |
| --- |
| create table t\_lock ( `id` int primary key auto\_increment, `name` varchar(50) ); create table t\_key ( `id` int primary key auto\_increment, `name` varchar(50),  `lock\_id` int, foreign key (`lock\_id`) references t\_lock (`id`) ); insert into t\_lock(`name`) values('阿里巴巴'); insert into t\_lock(`name`) values('华为'); insert into t\_lock(`name`) values('联想'); insert into t\_key(`name`,`lock\_id`) values('马云',1); insert into t\_key(`name`,`lock\_id`) values('任正非',2); insert into t\_key(`name`,`lock\_id`) values('柳传志',3); |

JavaBean

|  |
| --- |
| public class Key {  private Integer id;  private String name;  private Lock lock; } public class Lock {  private Integer id;  private String name; } |

### 根据id查询

KeyMapper.java

|  |
| --- |
| public interface KeyMapper {  Key queryKeyById(Integer id); } |

KeyMapper.xml

|  |
| --- |
| <resultMap id="queryKeyById\_resultMap" type="com.onetoone.pojo.Key">  <id column="id" property="id"/>  <result column="name" property="name"/>  <!-- 级联映射：因为key中含有lock对象，则可以通过lock.属性名的方式获取到lock对象的属性 -->  <result column="lock\_id" property="lock.id"/>  <result column="lock\_name" property="lock.name"/> </resultMap> <select id="queryKeyById" resultMap="queryKeyById\_resultMap">  select t\_key.\*, t\_lock.name lock\_name from t\_key,t\_lock where t\_key.lock\_id = t\_lock.id and t\_key.id = #{id} </select> |

### 单步查询

KeyMapper.java and LockMapper.java

|  |
| --- |
| public interface KeyMapper {  Key queryKeyById41Step(Integer id); } public interface LockMapper {  Lock queryLockById(Integer id); } |

KeyMapper.xml

|  |
| --- |
| <resultMap id="queryKeyById41Step\_resultMap" type="com.onetoone.pojo.Key">  <id column="id" property="id"/>  <id column="name" property="name"/>  <association property="lock" select="com.onetoone.mapper.LockMapper.queryLockById" column="lock\_id"/> </resultMap> <select id="queryKeyById41Step" resultMap="queryKeyById41Step\_resultMap">  select t\_key.\*, t\_lock.name lock\_name from t\_key,t\_lock where t\_key.lock\_id = t\_lock.id and t\_key.id = #{id} </select> |

LockMapper.xml

|  |
| --- |
| <select id="queryLockById" resultType="com.onetoone.pojo.Lock">  select id,name from t\_lock where id = #{id} </select> |

### 多步查询

KeyMapper.java and LockMapper.java

|  |
| --- |
| public interface KeyMapper {  Key queryKeyById42Step(Integer id); } public interface LockMapper {  Lock queryLockById(Integer id); } |

KeyMapper.xml

|  |
| --- |
| <resultMap id="queryKeyById42Step\_resultMap" type="com.onetoone.pojo.Key">  <id column="id" property="id"/>  <id column="name" property="name"/>  <association property="lock" select="com.onetoone.mapper.LockMapper.queryLockById" column="lock\_id"/> </resultMap> <select id="queryKeyById42Step" resultMap="queryKeyById42Step\_resultMap">  select id ,name,lock\_id from t\_key where id = #{id} </select> |

### 延时加载

对于多步查询的情况下，经常会需要配合延时加载一起使用。延迟加载在一定程度上可以减少很多没有必要的查询，给数据库和服务器提升性能上的优化。如果要启用延迟加载，需要在mybatis-config.xml配置文件中添加全局的settings配置。

|  |
| --- |
| <settings>  <!-- 打开延迟加载的开关 -->  <setting name="lazyLoadingEnabled" value="true" />  <!-- 将积极加载改为消极加载，按需加载 -->  <setting name="aggressiveLazyLoading" value="false"/> </settings> |

在mybatis以前的版本中，如果要完成延时加载，还需要导入cglib和asm的jar包，但是最新的mybatis已经内置了这两个jar包的功能，因此不再需要导入jar包而可以直接使用。

## 一对多

|  |
| --- |
| create table t\_clazz( `id` int primary key auto\_increment, `name` varchar(50)); insert into t\_clazz(`name`) values('javaEE20170228'); insert into t\_clazz(`name`) values('javaEE20170325'); insert into t\_clazz(`name`) values('javaEE20170420'); insert into t\_clazz(`name`) values('javaEE20170515'); create table t\_student( `id` int primary key auto\_increment, `name` varchar(50), `clazz\_id` int, foreign key(`clazz\_id`) references t\_clazz(`id`)); insert into t\_student(`name`,`clazz\_id`) values('stu0228\_1',1); insert into t\_student(`name`,`clazz\_id`) values('stu0228\_2',1); insert into t\_student(`name`,`clazz\_id`) values('stu0228\_3',1); insert into t\_student(`name`,`clazz\_id`) values('stu0325\_1',2); insert into t\_student(`name`,`clazz\_id`) values('stu0325\_2',2); insert into t\_student(`name`,`clazz\_id`) values('stu0420\_1',3); |

**JavaBean**

|  |
| --- |
| public class Clazz {  private Integer id;  private String name;  private List<Student> students; } public class Student {  private Integer id;  private String name;  private Clazz clazz;//双向关联 } |

### 单步查询

ClazzMapper.java

|  |
| --- |
| public interface ClazzMapper {  Clazz queryClazzById41Step(Integer id); } |

**ClazzMapper.xml**

|  |
| --- |
| <resultMap id="queryClazzById41Step\_resultMap" type="com.onetomany.pojo.Clazz">  <id column="id" property="id"/>  <result column="name" property="name"/>  <collection property="students" ofType="com.onetomany.pojo.Student">  <id column="stu\_id" property="id"/>  <result column="stu\_name" property="name"/>  </collection> </resultMap> <select id="queryClazzById41Step" resultMap="queryClazzById41Step\_resultMap">  select t\_clazz.\*, t\_student.id stu\_id, t\_student.name stu\_name  from t\_clazz left join t\_student on t\_clazz.id = t\_student.clazz\_id  where t\_clazz.id = #{id} </select> |

### 多步查询

ClaszzMapper.java and StudentMapper.java

|  |
| --- |
| public interface ClazzMapper {  Clazz queryClazzById42Step(Integer id); } public interface StudentMapper {  List<Student> queryStudentsByClazzId(Integer clazzId); } |

ClazzMapper.xml

|  |
| --- |
| <resultMap id="queryClazzById42Step\_resultMap" type="com.onetomany.pojo.Clazz">  <id column="id" property="id"/>  <result column="name" property="name"/>  <collection select="com.onetomany.mapper.StudentMapper.queryStudentsByClazzId" property="students" column="id"/> </resultMap> <select id="queryClazzById42Step" resultMap="queryClazzById42Step\_resultMap">  select id,name from t\_clazz where id = #{id} </select> |

StudentMapper.xml

|  |
| --- |
| <select id="queryStudentsByClazzId" resultType="com.onetomany.pojo.Student">  select id , name from t\_student where clazz\_id = #{clazzId} </select> |

### 双向关联

如果两个JavaBean中的互相包含对方的对象，就会形成双向关联。

ClazzMapper.xml

|  |
| --- |
| <resultMap id="queryClazzById42Step\_resultMap" type="com.onetomany.pojo.Clazz">  <id column="id" property="id"/>  <result column="name" property="name"/>  <collection select="com.onetomany.mapper.StudentMapper.queryStudentsByClazzId42Step" property="students" column="id"/> </resultMap> <select id="queryClazzById42Step" resultMap="queryClazzById42Step\_resultMap">  select id,name from t\_clazz where id = #{id} </select> |

StudentMapper.xml

|  |
| --- |
| <resultMap id="queryStudentsByClazzId42Step\_resultMap" type="com.onetomany.pojo.Student">  <id column="id" property="id"/>  <result column="name" property="name"/>  <association select="com.onetomany.mapper.ClazzMapper.queryClazzById42Step" property="clazz" column="clazz\_id"/> </resultMap> <select id="queryStudentsByClazzId42Step" resultMap="queryStudentsByClazzId42Step\_resultMap">  select id , name , clazz\_id from t\_student where clazz\_id = #{clazzId} </select> |

双向关联有时会出现死循环，可以从两个角度去避免这个情况：不要调用JavaBean的toString方法和在最后一次查询处，使用resultType封装对象。

# MyBatis动态SQL语句

## if标签

if语句可以动态的根据传入的值来决定是否需要动态的添加查询条件。

UserMapper.java

|  |
| --- |
| public interface UserMapper {  /\* 根据lastName和sex查询用户 \*/  public List<User> queryUsersByNameOrSex(User user); } |

UserMapper.xml

|  |
| --- |
| <select id="queryUsersByNameOrSex" resultType="com.sql.pojo.User">  select id,last\_name lastName,sex from  t\_user  <!-- if标签做if判断，判断User参数lastName属性值不为null -->  <if test="lastName != null">  last\_name like concat('%',#{lastName},'%') or  </if>  <if test="sex == 1 or sex == 0">  sex = #{sex}  </if> </select> |

## where标签

where语句可以在动态sql语句中有效的去除判断条件**前面**的and和or，但是其不能去除判断条件后面的and或or，必须使用trim标签。where标签还有一个作用就是可以动态的为sql语句添加where关键字，如果有查询条件就添加where关键字，如果没有查询条件就不添加where关键字。

UserMapper.java

|  |
| --- |
| public interface UserMapper {  /\* 根据lastName和sex查询用户 \*/  public List<User> queryUsersByNameOrSex(User user); } |

UserMapper.xml

|  |
| --- |
| <select id="queryUsersByNameOrSex" resultType="com.sql.pojo.User">  select id,last\_name lastName,sex from t\_user  <where>  <if test="lastName != null">  last\_name like concat('%',#{lastName},'%')  </if>  <if test="sex == 1 or sex == 0">  or sex = #{sex}  </if>  </where> </select> |

## trim标签

trim标签可以动态的在包含语句前面或后面添加内容，也可以去除前面或者后面的内容。

prefix属性 前面添加内容

suffix属性 后面添加内容

prefixOverrides属性 前面去除内容

suffixOverrides属性 后面去除内容

UserMapper.java

|  |
| --- |
| public interface UserMapper {  /\* 根据lastName和sex查询用户 \*/  public List<User> queryUsersByNameOrSex(User user); } |

UserMapper.xml

|  |
| --- |
| <select id="queryUsersByNameOrSex" resultType="com.sql.pojo.User">  select id,last\_name lastName,sex from t\_user  <trim prefixOverrides="or" suffixOverrides="or" prefix="where">  <if test="lastName != null">  last\_name like concat('%',#{lastName},'%')  </if>  <if test="sex == 1 or sex == 0">  or sex = #{sex}  </if>  </trim> </select> |

## choose标签

choose when otherwise可以执行多路选择判断，但是只有一个分支会被执行。类似于switch case语句。

UserMapper.java

|  |
| --- |
| public interface UserMapper {  /\*根据用户的lastName和sex查询用户信息，  如果lastName值有效的时候，只使用lastName查询条件查询  如果lastName值无效，则只使用sex来查询，  如果lastName和sex值都无效，则可以使用自定义查询条件。 \*/  public List<User> queryUsersByUser(User user); } |

UserMapper.xml

|  |
| --- |
| <select id="queryUsersByUser" resultType="com.sql.pojo.User">  select id,last\_name lastName,sex from t\_user where  <choose>  <when test="lastName != null">  last\_name like *concat*('%',#{lastName},'%')  </when>  <when test="sex == 1 or sex == 0">  sex = #{sex}  </when>  <otherwise>  sex = 1  </otherwise>  </choose> </select> |

## set标签

set标签可以用于删除条件后面的逗号，一般会和update语句一起使用，该功能也可以使用trim标签来实现。set标签还有一个功能就是可以用于添加set关键字，类似于where标签的功能。

UserMapper.java

|  |
| --- |
| public interface UserMapper {  /\* 更新用户，如果lastName值有效则更新last\_name，如果sex值有效，则更新sex \*/  public int updateUser(User user); } |

UserMapper.xml

|  |
| --- |
| <update id="updateUser" parameterType="com.sql.pojo.User">  update t\_user  <!-- set标签可以删除包含内容后面的逗号，并且还可以添加一set关键字 -->  <set>  <if test="lastName != null">  last\_name = #{lastName} ,  </if>  <if test="sex == 1 or sex == 0">  sex = #{sex}  </if>  </set>  where id = #{id} </update> |

## foreach标签

foreach标签可以用来遍历数据，一般会和sql语句中的in关键字配合使用。

collection属性 设置需要遍历的数据源

item属性 当前遍历的数据

separator属性 给遍历输出的每个元素中间添加的内容

open属性 遍历开始的时候输出的内容

close属性 遍历结束的时候输出的内容

UserMapper.java

|  |
| --- |
| public interface UserMapper {  /\*根据传递进来的id值遍历输出查询，select \* from 表名 where id in ( 1 , 2, 6 , 4 ); \*/  public List<User> queryUsersByIds(List<Integer> ids); } |

UserMapper.xml

|  |
| --- |
| <select id="queryUsersByIds" resultType="com.sql.pojo.User">  select id,last\_name lastName,sex from t\_user where id in  <!-- select \* from 表名 where id in ( 1 , 2, 6 , 4 ); -->  <foreach collection="list" item="i" separator="," open="(" close=")">  #{i}  </foreach> </select> |

# MyBatis缓存

缓存指的是把一些常用的数据，保存到一个可以高速读取的缓冲区中。方便程序在频繁读取的时候，可以快速的取出数据。这就叫做缓存。

Mybatis中的缓存分为一级缓存和二级缓存，其一级缓存指的是将数据保存到SqlSession中，同一个SqlSession对此sql操作都可以获取。其二级缓存指的是将数据保存到SqlSessionFactory中，多个SqlSession对象可以共享这些数据。

## 一级缓存

每次执行查询时，mybatis都会到SqlSession中查看是否有该数据，如果有就直接返回而不到数据库中查询；如果没有就发送sql语句到数据库中查询并将返回的查询结果保存到SqlSession缓存中。

### 示例

|  |
| --- |
| public void testQueryUserById() {  SqlSession session = sqlSessionFactory.openSession();  try {  UserMapper mapper = session.getMapper(UserMapper.class);  User user = mapper.queryUserById(1);  System.*out*.println(user);  //第二次查询并没有执行发送sql语句到数据库中，而是直接返回了一级缓存中的数据  user = mapper.queryUserById(1);  System.*out*.println(user);  } finally {  session.close();  } } |

|  |
| --- |
| <select id="queryUserById" resultType="com.cache.pojo.User">  select id,last\_name lastName,sex from t\_user where id = #{id} </select> |

### 缓存失效

一级缓存如果遇到下列四种情况时就会失效：

1、不在同一个SqlSession对象中

2、执行语句的参数不同，缓存中也就不存在数据

3、执行增删改语句后会清空掉缓存

4、手动清空缓存

#### 不在同一个SqlSession对象中

|  |
| --- |
| public void testCacheFail1() throws Exception {  SqlSession session1 = sqlSessionFactory.openSession();  UserMapper mapper1 = session1.getMapper(UserMapper.class);  System.*out*.println(mapper1.queryUserById(1));  session1.close();  //因为前后两次查询不在同一个SqlSession对象中，因此不再同一个缓存中  SqlSession session2 = sqlSessionFactory.openSession();  UserMapper mapper2 = session2.getMapper(UserMapper.class);  System.*out*.println(mapper2.queryUserById(1));  session2.close(); } |

#### 执行语句的参数不同，缓存中也就不存在数据

|  |
| --- |
| public void testCacheFail2() throws Exception {  SqlSession session1 = sqlSessionFactory.openSession();  UserMapper mapper1 = session1.getMapper(UserMapper.class);  System.*out*.println(mapper1.queryUserById(1));  System.*out*.println(mapper1.queryUserById(2));  session1.close(); } |

#### 执行增删改语句后会清空掉缓存

|  |
| --- |
| public void testCacheFail3() throws Exception {  SqlSession session1 = sqlSessionFactory.openSession();  UserMapper mapper1 = session1.getMapper(UserMapper.class);  System.*out*.println(mapper1.queryUserById(12));  // 当执行update、insert、delete语句的时候，会清空缓存  mapper1.updateUser(new User(4, "abc", 1));  session1.commit();  System.*out*.println(mapper1.queryUserById(12));  session1.close(); } |

#### 手动清空缓存

|  |
| --- |
| public void testCacheFail4() throws Exception {  SqlSession session1 = sqlSessionFactory.openSession();  UserMapper mapper1 = session1.getMapper(UserMapper.class);  System.*out*.println(mapper1.queryUserById(12));  // 通过调用SqlSession的clearCache方法清空缓存  session1.clearCache();  System.*out*.println(mapper1.queryUserById(12));  session1.close(); } |

## 二级缓存

在一级缓存SqlSession对象调用close方法关闭时，就会将其内部缓存的对象存储到二级缓存SqlSessionFactory中。

**二级缓存命中率**

可以根据二级缓存命中率的高低来判断哪些缓存被读取的更频繁，这样可以知道哪些数据可以缓存，哪些数据不需要缓存，从而提高效率。

### 使用注意

mybatis中二级缓存是默认不开启的，需要配置其核心配置文件中的settings配置

|  |
| --- |
| <settings>  <!-- 表示允许使用二级缓存，打开二缓存。 -->  <setting name="cacheEnabled" value="true"/> </settings> |

配置完核心配置文件后在Mapper.xml中加入cache缓存标签，表示使用缓存

|  |
| --- |
| <!-- 使用二级缓存 --> <cache/> |

被二级缓存的对象必须要实现Java的序列化接口

|  |
| --- |
| public class User implements Serializable {  private Integer id;  private String lastName;  private Integer sex; } |

**示例**

|  |
| --- |
| public void queryOne() {  SqlSession session = sqlSessionFactory.openSession();  UserMapper mapper = session.getMapper(UserMapper.class);  System.*out*.println(mapper.queryUserById(12));  session.close(); } @Test public void testSecondCache() throws Exception {  queryOne();  queryOne();  queryOne(); } |

### 相关属性标签

#### userCache属性

select标签的属性userCache用于设置当前查询是否使用二级缓存，默认值为true，表示使用二级缓存；设置为false表示不适用二级缓存。

|  |
| --- |
| <select id="queryUserById" resultType="com.cache.pojo.User" useCache="true">  select id,last\_name lastName,sex from t\_user where id = #{id} </select> |

#### flushCache属性

update|insert|delete标签的属性flushCache用于设置是否清空缓存，默认值为true，表示执行当前语句后清空缓存；设置为false表示执行完当前语句后不清空缓存。当flushCache设置为false时，就会出现脏读的情况。

|  |
| --- |
| <update id="updateUser" parameterType="com.cache.pojo.User" flushCache="false">  update t\_user set last\_name = #{lastName},sex = #{sex} where id = #{id} </update> |

#### cache标签

默认**cache**标签的作用如下：

1、mapper.xml文件中所有的**select**语句都将会被缓存

2、mapper.xml文件中所有的**insert**|**update**|**delete**都会刷新缓存

3、缓存会使用Least Recently Used(LRU最近最少使用)算法来收回

4、根据时间表(比如 no Flush Interval,没有刷新间隔), 缓存不会以任何时间顺序来刷新。

5、缓存会存储列表集合或对象(无论查询方法返回什么)的 1024 个引用。

6、缓存会被视为是 read/write(可读/可写)的缓存,对象检索不是共享的,可以安全地被调用者修改,而不干扰其他调用者或线程所做的潜在修改。

eviction属性 表示缓存策略

LRU 最近最少使用的:移除最长时间不被使用的对象(这是默认策略)。

FIFO 先进先出:按对象进入缓存的顺序来移除它们。

SOFT 软引用:移除基于垃圾回收器状态和软引用规则的对象。

WEAK 弱引用:更积极地移除基于垃圾收集器状态和弱引用规则的对象。

flushInterval属性 表示间隔多长时间刷新缓冲区，清理溢出数据，以毫秒为单位。

size属性 表示缓存中可以保存多少个对象，默认是1024。

readOnly属性 表示是否只读。

true 表示缓存中只有一个对象。

false（默认） 每次取出来都会反序列化拷贝一份。

type属性 表示自定义二级缓存对象。

## 缓存的顺序

1. 执行查询语句时，mybatis先在二级缓存中查询数据，如果二级缓存中没有，再到一级缓存中查找。
2. 如果二级缓存和一级缓存中都没有数据，那么就会发送sql语句到数据库中查询。
3. 将查询出来的结果保存到一级缓存中。
4. 当SqlSession关闭，即一级缓存关闭时，会将一级缓存中的数据保存到二级缓存中。

## 自定义缓存

mybatis中的缓存本质上是一个HashMap集合，且该Map集合的键值对都是Object类型。可以通过实现Cache接口来自定义缓存。

# MyBatis逆向工程

mybatis逆向工程，简称MBG，是专门为mybatis框架使用者定制的代码生成器，可以快速的根据数据库表生成Mapper.xml配置文件、Mapper接口以及JavaBean类。

mybatis逆向工程需要一个可以自动对数据库表生成CRUD代码的插件，即mybatis-generator-core.jar。其可以对比数据库表后生成大量的基础代码。

## 逆向工程配置

**commentGenerator标签**

该标签的作用是配置自动生成代码时是否自动生成注释，其value属性为true时表示不生成注释。

## 逆向工程示例

配置文件mbg.xml

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE generatorConfiguration PUBLIC "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN" "http://mybatis.org/dtd/mybatis-generator-config\_1\_0.dtd"> <generatorConfiguration>  <!--targetRuntime 属性设置你生成的版本  MyBatis3 豪华版  MyBatis3Simple 标准版本 CRUD -->  <context id="MySQLTables" targetRuntime="MyBatis3Simple">  <!-- 去掉全部的注释 -->  <commentGenerator>  <property name="suppressAllComments" value="true" />  </commentGenerator>  <!-- 修改数据库的四个连接属性 -->  <jdbcConnection driverClass="com.mysql.jdbc.Driver"  connectionURL="jdbc:mysql://localhost:3306/mbg"  userId="root"  password="root">  </jdbcConnection>  <javaTypeResolver >  <property name="forceBigDecimals" value="false" />  </javaTypeResolver>  <!-- javaModelGenerator 配置生成javaBean  targetPackage 生成的javaBean包名  targetProject 修改哪个项目（路径） => targetProject=".\src" -->  <javaModelGenerator targetPackage="com.mbg.pojo" targetProject="day611/src">  <property name="enableSubPackages" value="true" />  <property name="trimStrings" value="true" />  </javaModelGenerator>  <!-- sqlMapGenerator 配置生成SQL的Mapper.xml配置文件 -->  <sqlMapGenerator targetPackage="com.mbg.mapper" targetProject="day611/src">  <property name="enableSubPackages" value="true" />  </sqlMapGenerator>  <!-- javaClientGenerator 配置生成Mapper接口 -->  <javaClientGenerator type="XMLMAPPER" targetPackage="com.mbg.mapper" targetProject="day611/src">  <property name="enableSubPackages" value="true" />  </javaClientGenerator>  <!-- 一个表，对应一个table标签  tableName 数据库表名  domainObjectName 生成对应的javaBean的类名 -->  <table tableName="t\_user" domainObjectName="User" />  <table tableName="t\_book" domainObjectName="Book" />  </context> </generatorConfiguration> |

Java代码

|  |
| --- |
| public static void main(String[] args) throws Exception {  List<String> warnings = new ArrayList<String>();  boolean overwrite = true;  File configFile = new File("day611/config/mbg.xml");  ConfigurationParser cp = new ConfigurationParser(warnings);  Configuration config = cp.parseConfiguration(configFile);  DefaultShellCallback callback = new DefaultShellCallback(overwrite);  MyBatisGenerator myBatisGenerator = new MyBatisGenerator(config,  callback, warnings);  myBatisGenerator.generate(null); } |

# 通用Mapper

通用Mapper可以通过Mybatis的拦截器原理，动态的实现单表的增删改查功能，大大降低了开发成本，减少工作量。

该Mapper是一个接口，里面提供给了一系列的方法，导入依赖，配置插件，利用Mybatis使用通用的Mapper接口，首先继承该接口，泛型为类中的类型。这样就没有配置文件了，所以要在Mybatis核心配置文件中，配置扫描的包。

## 相关依赖

**非SpringBoot**

|  |
| --- |
| <dependency>  <groupId>com.github.abel533</groupId>  <artifactId>mapper</artifactId>  <version>2.3.4</version> </dependency> |

|  |
| --- |
| <!-- 通用Mapper的插件 --> <plugins>  <plugin interceptor="com.github.abel533.mapperhelper.MapperInterceptor">  <!--主键自增回写方法,默认值MYSQL -->  <property name="IDENTITY" value="MYSQL" />  <!--通用Mapper默认接口，我们定义的Mapper需要实现该接口 -->  <property name="mappers" value="com.github.abel533.mapper.Mapper" />  </plugin> </plugins> |

**SpringBoot**

|  |
| --- |
| <dependency>  <groupId>tk.mybatis</groupId>  <artifactId>mapper-spring-boot-starter</artifactId>  <version>1.2.3</version>  <exclusions>  <exclusion>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-jdbc</artifactId>  </exclusion>  </exclusions> </dependency> |

## 通用Mapper接口API

|  |  |
| --- | --- |
| **通用Mapper接口API** | |
| List<T> | **select**(T record) 根据实体中的属性值进行查询，查询条件使用等号 |
| T | **selectByPrimaryKey**(Object key) 根据主键字段进行查询，方法参数必须包含完整的主键属性，查询条件使用等号 |
| List<T> | **selectAll**()查询全部结果，select(null)方法能达到同样的效果 |
| T | **selectOne**(T record) 根据实体中的属性进行查询，只能有一个返回值，有多个结果是抛出异常，查询条件使用等号 |
| int | **selectCount**(T record)根据实体中的属性查询总数，查询条件使用等号 |
| int | **insert**(T record) 保存一个实体，null的属性也会保存，不会使用数据库默认值 |
| int | **insertSelective**(T record) 保存一个实体，null的属性不会保存，会使用数据库默认值 |
| int | **updateByPrimaryKey**(T record) 根据主键更新实体全部字段，null值会被更新 |
| int | **updateByPrimaryKeySelective**(T record) 根据主键更新属性不为null的值 |
| int | **delete**(T record) 根据实体属性作为条件进行删除，查询条件使用等号 |
| int | **deleteByPrimaryKey**(Object key) 根据主键字段进行删除，方法参数必须包含完整的主键属性 |
| List<T> | **selectByExample**(Object example)根据Example条件进行查询，这个查询支持通过Example类指定查询列，通过selectProperties方法指定查询列 |
| int | **selectCountByExample**(Object example) 根据Example条件进行查询总数 |
| int | **updateByExample**(@Param("record") T record, @Param("example") Object example) 根据Example条件更新实体record包含的全部属性，null值会被更新 |
| int | **updateByExampleSelective**(@Param("record") T record, @Param("example") Object example)根据Example条件更新实体record包含的不是null的属性值 |
| int | **deleteByExample**(Object example)根据Example条件删除数据 |

# MyBatis扩展

## 插入记录返回主键

要实现往数据库中添加数据后返回主键信息的功能有三种方式，第一种是使用**insert**标签中的**useGenerateKeys**属性和**keyProperty**属性组合使用获取主键信息；第二种是使用子元素**selectKey**标签执行SQL语句获取；第三种是使用注解@SelectKey来获取插入数据的主键信息。

### 设置属性方式

|  |
| --- |
| <!--useGeneratedKeys设置为true表示返回生成的主键，keyProperty的值表示返回的主键值注入到哪个属性中--> <insert id="saveUser" parameterType="com.xmm.pojo.User" useGeneratedKeys="true" keyProperty="id">  insert into t\_user(`last\_name`,`sex`) values(#{lastName},#{sex}); </insert> |

### 设置标签方式

**selectKey**标签是专门用来查询生成的主键

**order**属性 **AFTER**插入之后执行；**BEFORE**插入之前执行

**keyProperty**属性 设置返回的主键注入的属性

**resultType**属性 设置返回的数据类型

|  |
| --- |
| <insert id="saveUser" parameterType="com.xmm.pojo.User">  <selectKey order="AFTER" keyProperty="id" resultType="int">  SELECT *LAST\_INSERT\_ID*()  </selectKey>  insert into t\_user(`last\_name`,`sex`) values(#{lastName},#{sex}); </insert> |

**补充**：selectKey标签在Oracle数据库的用法

|  |
| --- |
| <selectKey order="BEFORE" resultType="int" keyProperty="id">  select 序列名.nextval as id from dual </selectKey> |

### 设置注解方式

|  |
| --- |
| @SelectKey(statement = "select last\_insert\_id()", before = false, keyProperty = "id", resultType = Integer.class) @Insert("insert into t\_user(`last\_name`,`sex`) values(#{lastName},#{sex})") public int saveUser(User user); |

**测试方法**：

|  |
| --- |
| public void testSaveUser() {  SqlSession session = sqlSessionFactory.openSession();  try {  UserMapper mapper = session.getMapper(UserMapper.class);  User user = new User(null, "user", 0);  mapper.saveUser(user);  System.*out*.println(user.getId());  session.commit();  } finally {  session.close();  } } |

## EnCache缓存的使用

EhCache 是专业的纯Java进程缓存框架。具有快速、精干等特点，是Hibernate中默认的CacheProvider缓存提供者。由于MyBatis中定义了**Cache**缓存接口极大的方便了我们对缓存的自定义切换和扩展。

### encache使用步骤

1. 使用EnCache首先要导入如下的jar包：encache-core.jar | mybatis-encache.jar | slf4j-api | slf4j-log4j12.jar。

2、然后要配置encache.xml配置文件

*<?***xml version="1.0" encoding="UTF-8"***?>*<**ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:noNamespaceSchemaLocation="../config/ehcache.xsd"**>  
 <!-- 磁盘保存路径 -->  
 <**diskStore path="D:\ehcache"** />  
  
 <**defaultCache  
 maxElementsInMemory="1000"  
 maxElementsOnDisk="10000000"  
 eternal="false"  
 overflowToDisk="true"  
 timeToIdleSeconds="120"  
 timeToLiveSeconds="120"  
 diskExpiryThreadIntervalSeconds="120"  
 memoryStoreEvictionPolicy="LRU"**>  
 </**defaultCache**>  
</**ehcache**>

diskStore：指定数据在磁盘中的存储位置。

以下属性是必须的：

maxElementsInMemory - 在内存中缓存的element的最大数目

maxElementsOnDisk - 在磁盘上缓存的element的最大数目，若是0表示无穷大

eternal - 设定缓存的elements是否永远不过期。如果为true，则缓存的数据始终有效，如果为false那么还要根据timeToIdleSeconds，timeToLiveSeconds判断

overflowToDisk - 设定当内存缓存溢出的时候是否将过期的element缓存到磁盘上

以下属性是可选的：

timeToIdleSeconds - 当缓存在EhCache中的数据前后两次访问的时间超过timeToIdleSeconds的属性取值时，这些数据便会删除，默认值是0,也就是可闲置时间无穷大

timeToLiveSeconds - 缓存element的有效生命期，默认是0.,也就是element存活时间无穷大

diskSpoolBufferSizeMB 这个参数设置DiskStore(磁盘缓存)的缓存区大小.默认是30MB.每个Cache都应该有自己的一个缓冲区.

diskPersistent - 在VM重启的时候是否启用磁盘保存EhCache中的数据，默认是false。

diskExpiryThreadIntervalSeconds - 磁盘缓存的清理线程运行间隔，默认是120秒。每个120s，相应的线程会进行一次EhCache中数据的清理工作

memoryStoreEvictionPolicy - 当内存缓存达到最大，有新的element加入的时候， 移除缓存中element的策略。默认是LRU（最近最少使用），可选的有LFU（最不常使用）和FIFO（先进先出）

3、配置mybatis的**cache**标签的**type**属性

<**cache type="org.mybatis.caches.ehcache.EhcacheCache"**></**cache**>

## 分页插件PagaHelper

1. 导入mybatis-pageHelper的jar包：jsqlparser.jar | pagehelper.jar
2. 在mybatis-config.xml核心配置文件中配置分页的拼接器

<**plugins**>  
 <!-- com.github.pagehelper为PageHelper类所在包名 -->  
 <**plugin interceptor="com.github.pagehelper.PageInterceptor"**></**plugin**>  
</**plugins**>

1. 使用mybatis-pageHelper代码示例

**public void** testQueryUsers() {  
 SqlSession session = sqlSessionFactory.openSession();  
 **try** {  
 Page<User> page = PageHelper.startPage(1, 5);  
 UserMapper userMapper = session.getMapper(UserMapper.**class**);  
 List<User> list = userMapper.queryUsers();  
 System.**out**.println("当前页：" + page.getPageNum());  
 System.**out**.println("总页数：" + page.getPages());  
 System.**out**.println("每页显示数量：" + page.getPageSize());  
 System.**out**.println("总记录数：" + page.getTotal());  
 System.**out**.println("当前页数据：" + page.getResult());  
 System.**out**.println(page);  
 **for** (User user : list) {  
 System.**out**.println(user);  
 }  
 } **finally** {  
 session.close();  
 }  
}

注：暂缺UserMapper.xml配置文件配置queryUsers方法。

### PageInfo类

该类是专门给分页使用的类，用于计算需要显示的页码。

**public void** testPageInfoQueryUsers() {  
 SqlSession session = sqlSessionFactory.openSession();  
 **try** {  
 UserMapper userMapper = session.getMapper(UserMapper.**class**);  
 PageHelper.startPage(5, 2);  
 List<User> list = userMapper.queryUsers();  
 PageInfo<User> pageInfo = **new** PageInfo<>(list, 5);  
 System.**out**.println("当前页：" + pageInfo.getPageNum());  
 System.**out**.println("总页数：" + pageInfo.getPages());  
 System.**out**.println("每页显示数量：" + pageInfo.getPageSize());  
 System.**out**.println("总记录数：" + pageInfo.getTotal());  
 // 遍历当前页数据  
 **for** (User user : list) {  
 System.**out**.println(user);  
 }  
 // 遍历分页条页码  
 System.**out**.print("分页条，页码是：");  
 **for** (**int** i : pageInfo.getNavigatepageNums() ) {  
 System.**out**.print( i + "**\t**" );  
 }  
 System.**out**.println();  
 } **finally** {  
 session.close();  
 }  
}

# MyBatis Plus

MyBatis-Plus（简称 MP）是一个 MyBatis 的增强工具，在 MyBatis 的基础上只做增强不做改变，为简化开发、提高效率而生。

**无侵入**：只做增强不做改变，引入它不会对现有工程产生影响，如丝般顺滑

**损耗小**：启动即会自动注入基本 CURD，性能基本无损耗，直接面向对象操作

**强大的 CRUD 操作**：内置通用 Mapper、通用 Service，仅仅通过少量配置即可实现单表大部分 CRUD 操作，更有强大的条件构造器，满足各类使用需求

**支持 Lambda 形式调用**：通过 Lambda 表达式，方便的编写各类查询条件，无需再担心字段写错

**支持多种数据库**：支持 MySQL、MariaDB、Oracle、DB2、H2、HSQL、SQLite、Postgre、SQLServer2005、SQLServer 等多种数据库

**支持主键自动生成**：支持多达 4 种主键策略（内含分布式唯一 ID 生成器 - Sequence），可自由配置，完美解决主键问题

**支持 XML 热加载**：Mapper 对应的 XML 支持热加载，对于简单的 CRUD 操作，甚至可以无 XML 启动

**支持 ActiveRecord 模式**：支持 ActiveRecord 形式调用，实体类只需继承 Model 类即可进行强大的 CRUD 操作

**支持自定义全局通用操作**：支持全局通用方法注入（ Write once, use anywhere ）

**支持关键词自动转义**：支持数据库关键词（order、key......）自动转义，还可自定义关键词

**内置代码生成器**：采用代码或者 Maven 插件可快速生成 Mapper 、 Model 、 Service 、 Controller 层代码，支持模板引擎，更有超多自定义配置等您来使用

**内置分页插件**：基于 MyBatis 物理分页，开发者无需关心具体操作，配置好插件之后，写分页等同于普通 List 查询

**内置性能分析插件**：可输出 Sql 语句以及其执行时间，建议开发测试时启用该功能，能快速揪出慢查询

**内置全局拦截插件**：提供全表 delete 、 update 操作智能分析阻断，也可自定义拦截规则，预防误操作

**内置 Sql 注入剥离器**：支持 Sql 注入剥离，有效预防 Sql 注入攻击。

## MyBatis Plus入门

建立user数据库表并向表中添加测试数据，建表及添加数据SQL语句如下：

|  |
| --- |
| DROP TABLE IF EXISTS user; CREATE TABLE user (  id BIGINT(20) NOT NULL COMMENT '主键ID',  name VARCHAR(30) NULL DEFAULT NULL COMMENT '姓名',  age INT(11) NULL DEFAULT NULL COMMENT '年龄',  email VARCHAR(50) NULL DEFAULT NULL COMMENT '邮箱',  PRIMARY KEY (id) ); INSERT INTO user (id, name, age, email) VALUES (1, 'Jone', 18, 'test1@baomidou.com'), (2, 'Jack', 20, 'test2@baomidou.com'), (3, 'Tom', 28, 'test3@baomidou.com'), (4, 'Sandy', 21, 'test4@baomidou.com'), (5, 'Billie', 24, 'test5@baomidou.com'); |

创建测试项目并引入MyBatis Plus相关依赖

|  |
| --- |
| <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  <!--MyBatis Plus的依赖，引入该依赖后不要再引入MyBatis-Spring的依赖，避免版本冲突-->  <dependency>  <groupId>com.baomidou</groupId>  <artifactId>mybatis-plus-boot-starter</artifactId>  <version>3.0.5</version>  </dependency>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  </dependency>  <!--在IDEA中安装好Lombok插件后还要添加Lombok的依赖-->  <dependency>  <groupId>org.projectlombok</groupId>  <artifactId>lombok</artifactId>  <optional>true</optional>  </dependency> </dependencies> |

配置Spring Boot测试项目中关于MySQL数据库相关的配置。

|  |
| --- |
| #MySQL数据库连接 spring.datasource.driver-class-name=com.mysql.jdbc.Driver spring.datasource.url=jdbc:mysql://localhost:3306/mp?useSSL=false spring.datasource.username=root spring.datasource.password=root |

编写Spring Boot测试项目的主启动类。

|  |
| --- |
| @SpringBootApplication public class MpDemoApplication {  public static void main(String[] args) {  SpringApplication.*run*(MpDemoApplication.class, args);  } } |

根据创建的测试数据库表的字段创建对应的JavaBean类，并使用Lombok注解来避免编写繁琐的getter、setter、constructor和toString等方法。

|  |
| --- |
| @Data public class User {  private Long id;  private String name;  private Integer age;  private String email; } |

编写通用Mapper接口，该接口需要继承MyBatis Plus中的BaseMapper接口，然后添加@Component注解使Spring容器能够扫描到该Mapper接口。

|  |
| --- |
| @Component public interface UserMapper extends BaseMapper<User> { } |

测试使用MyBatis Plus。

|  |
| --- |
| @RunWith(SpringRunner.class) @SpringBootTest public class MpDemoApplicationTests {  @Autowired  UserMapper userMapper;  @Test  public void selectList(){  // selectList()方法的参数是MP内置的条件封装器Wrapper  // 因为此时不需要查询条件，所以为null  List<User> users = userMapper.selectList(null);  users.forEach(System.*err*::println);  }  } |

为了方便开发，可以在Spring Boot的配置文件中配置MP的SQL输出日志，这样就能在控制台中看到执行的SQL语句。

|  |
| --- |
| #Mybatis日志 mybatis-plus.configuration.log-impl=org.apache.ibatis.logging.stdout.StdOutImpl |

## MyBatis Plus的CRUD

### insert插入操作

MyBatis Plus的插入操作如下，通过通用Mapper的insert方法插入一条数据，在MyBatis Plus中插入数据并不需要设置主键id，因为MyBatis Plus含有一个主键策略，可以自动生成id。

|  |
| --- |
| @Test public void insertUser() {  User user = new User();  user.setName("Tom");  user.setEmail("tom@qq.com");  user.setAge(19);  // insert方法会返回一个int类型的数值，表示SQL语句影响的行数  userMapper.insert(user); } |

#### 主键策略

MyBatis Plus的主键策略有五种形式自动填充插入数据的id，其中ID\_WORKER、UUID和ID\_WORKER\_STR只有当插入对象id为空时才会自动填充。五种主键策略如下:

|  |
| --- |
| public enum IdType {  // 自增长，需要在创建数据库表时设置主键自增  AUTO(0),  // 不设置规则，未设置主键类型  NONE(1),  // 手动输入的  INPUT(2),  // ID\_WORKER（Long类型）  ID\_WORKER(3),  // UUID  UUID(4),  // ID\_WORKER\_STR（idworker的字符串表示）  ID\_WORKER\_STR(5);  private int key;  private IdType(int key) {  this.key = key;  }  public int getKey() {  return this.key;  } } |

在对应的JavaBean的id属性上使用@TableId注解并设置需要使用的主键策略，默认是使用ID\_WORKER。

|  |
| --- |
| @TableId(type = IdType.*ID\_WORKER*) private Long id; |

如果要影响所有JavaBean的主键id的配置，可以在Spring Boot的配置文件中进行如下配置：

|  |
| --- |
| #全局设置主键生成策略 mybatis-plus.global-config.db-config.id-type=*auto* |

#### IdWorker

MyBatis Plus中自动填充的主键策略ID\_WORKER实现方式如下：

|  |
| --- |
| /\* 分布式自增长ID  \* 核心代码为其IdWorker类的实现，其原理结构如下，用一个"0"表示一位，用"—"分割：  \* 1||0---0000000000 0000000000 0000000000 0000000000 0 ---00000 ---00000 ---000000000000  \* 在上述字符串中，64位ID = (42(毫秒)+5(机器ID)+5(业务编码)+12(重复累加))  \* 1 第一位为未使用（或者作为long的符号位），  \* 2-42 41位的毫秒级时间，  \* 43-52 5位datacenter标识位，5位机器ID（并不算标识符，实际是为线程标识），  \* 53-64 12位该毫秒内的当前毫秒内的计数，加起来刚好64位，为一个Long型  \* 整体上按照时间自增排序，并且整个分布式系统内不会产生ID碰撞（由datacenter和机器ID作区分）  \* 效率较高，每秒能够产生26万ID左右，完全满足需要。\*/ public class IdWorker {  // 时间起始标记点，作为基准，一般取系统的最近时间（一旦确定不能变动）  private final static long *twepoch* = 1288834974657L;  // 机器标识位数  private final static long *workerIdBits* = 5L;  // 数据中心标识位数  private final static long *datacenterIdBits* = 5L;  // 机器ID最大值  private final static long *maxWorkerId* = -1L ^ (-1L << *workerIdBits*);  // 数据中心ID最大值  private final static long *maxDatacenterId* = -1L ^ (-1L << *datacenterIdBits*);  // 毫秒内自增位  private final static long *sequenceBits* = 12L;  // 机器ID偏左移12位  private final static long *workerIdShift* = *sequenceBits*;  // 数据中心ID左移17位  private final static long *datacenterIdShift* = *sequenceBits* + *workerIdBits*;  // 时间毫秒左移22位  private final static long *timestampLeftShift* = *sequenceBits* + *workerIdBits* +  *datacenterIdBits*;  private final static long *sequenceMask* = -1L ^ (-1L << *sequenceBits*);  /\* 上次生产id时间戳 \*/  private static long *lastTimestamp* = -1L;  // 0，并发控制  private long sequence = 0L;  private final long workerId;  // 数据标识id部分  private final long datacenterId;   public IdWorker() {  this.datacenterId = *getDatacenterId*(*maxDatacenterId*);  this.workerId = *getMaxWorkerId*(datacenterId, *maxWorkerId*);  }   //@param workerId 工作机器ID  //@param datacenterId 序列号  public IdWorker(long workerId, long datacenterId) {  if (workerId > *maxWorkerId* || workerId < 0) {  throw new IllegalArgumentException(String.*format*(  "worker Id can't be greater than %d or less than 0",   *maxWorkerId*));  }  if (datacenterId > *maxDatacenterId* || datacenterId < 0) {  throw new IllegalArgumentException(String.*format*(  "datacenter Id can't be greater than %d or less than 0",   *maxDatacenterId*));  }  this.workerId = workerId;  this.datacenterId = datacenterId;  }   //获取下一个ID @return  public synchronized long nextId() {  long timestamp = timeGen();  if (timestamp < *lastTimestamp*) {  throw new RuntimeException(String.*format*(  "Clock moved backwards. Refusing to generate id for %d milliseconds",  *lastTimestamp* - timestamp));  }  if (*lastTimestamp* == timestamp) {  // 当前毫秒内，则+1  sequence = (sequence + 1) & *sequenceMask*;  if (sequence == 0) {  // 当前毫秒内计数满了，则等待下一秒  timestamp = tilNextMillis(*lastTimestamp*);  }  } else {  sequence = 0L;  }  *lastTimestamp* = timestamp;  // ID偏移组合生成最终的ID，并返回ID  long nextId = ((timestamp - *twepoch*) << *timestampLeftShift*)  | (datacenterId << *datacenterIdShift*)  | (workerId << *workerIdShift*) | sequence;  return nextId;  }   private long tilNextMillis(final long lastTimestamp) {  long timestamp = this.timeGen();  while (timestamp <= lastTimestamp) {  timestamp = this.timeGen();  }  return timestamp;  }   private long timeGen() {  return System.*currentTimeMillis*();  }   // 获取 maxWorkerId  protected static long getMaxWorkerId(long datacenterId, long maxWorkerId) {  StringBuffer mpid = new StringBuffer();  mpid.append(datacenterId);  String name = ManagementFactory.*getRuntimeMXBean*().getName();  if (!name.isEmpty()) {  //GET jvmPid  mpid.append(name.split("@")[0]);  }  //MAC + PID 的 hashcode 获取16个低位  return (mpid.toString().hashCode() & 0xffff) % (maxWorkerId + 1);  }   // 数据标识id部分  protected static long getDatacenterId(long maxDatacenterId) {  long id = 0L;  try {  InetAddress ip = InetAddress.*getLocalHost*();  NetworkInterface network = NetworkInterface.*getByInetAddress*(ip);  if (network == null) {  id = 1L;  } else {  byte[] mac = network.getHardwareAddress();  id = ((0x000000FF & (long) mac[mac.length - 1])  | (0x0000FF00 & (((long) mac[mac.length - 2]) << 8))) >> 6;  id = id % (maxDatacenterId + 1);  }  } catch (Exception e) {  System.*out*.println(" getDatacenterId: " + e.getMessage());  }  return id;  } } |

### delete删除操作

|  |
| --- |
| @Test public void deleteUserById(){  // 根据多个ID删除  userMapper.deleteBatchIds(Arrays.*asList*(  1167253002278584322L,  1167252359367335937L,  1167252359568662529L));  userMapper.deleteById(1167253002278584322L);  Map<String,Object> map = new HashMap<>();  map.put("name", "Tom");  map.put("age", 0);  userMapper.deleteByMap(map); } |

### update修改操作

MyBatis Plus的update修改操作时生成的动态SQL是update user set name=? where id=?，通用Mapper执行updateById方法返回一个int类型的数值，表示执行的SQL语句影响的数据行数。

|  |
| --- |
| @Test public void updateUserById(){  User user = new User();  user.setId(1167251854343761922L);  user.setName("Bob");  userMapper.updateById(user); } |

在进行数据的修改操作时需要注意保持数据的一致性，此时就需要通过设置乐观锁来避免数据不一致的问题。

一般实际开发中更新数据库都会有一个日志系统记录修改的信息，包括但不限于修改数据的用户、修改的命令、修改的时间和状态。

#### version乐观锁

悲观锁：认为任何操作都是非法的，所以每一个操作都给你加了锁 优点：安全； 缺点：效率低；

乐观锁：认为任何操作都是正常的，所以每一个操作都不会加锁 优点：效率高；缺点：不安全。

乐观锁主要适用于当要更新一条记录的时候，希望这条记录没有被别人更新，也就是说实现线程安全的数据更新的场景。其实现方式如下：

取出记录时，获取当前数据的version，更新操作时将这个version作为条件并+1，然后执行更新。如果另一个SQL语句获取的数据version是修改之前的version而不是修改之后的version时，此时执行更新就会失败，从而避免了数据不一致。

|  |
| --- |
| UPDATE USER SET `name`='demo1', `version`=`version` + 1 WHERE id=1 AND `version`=1; UPDATE USER SET `name`='demo2', `version`=`version` + 1 WHERE id=1 AND `version`=1; |

上述两个更新操作中，第一个更新执行完成时，version的值就会变成2，而执行第二个更新时`version`=1就会返回false，从而导致无法更新。

在user表中添加version字段

|  |
| --- |
| ALTER TABLE `user` ADD COLUMN `version` INT DEFAULT 1; |

在JavaBean中添加version属性，并使用@Version注解标注，然后配置version属性值在插入数据时自动填充。

|  |
| --- |
| // 该注解表示版本关键字，被乐观锁加载 @Version private Integer version; |

|  |
| --- |
| @Component public class DateMetaObjectHandler implements MetaObjectHandler {  @Override  public void insertFill(MetaObject metaObject) {  this.setFieldValByName("version", 0, metaObject);  } } |

创建配置类，并通过@Bean注解将乐观锁插件注册到Spring容器中。

1、支持的数据类型只有int,Integer,long,Long,Date,Timestamp,LocalDateTime；

2、整数类型下，newVersion = oldVersion + 1；

3、newVersion会回写到entity中；

4、仅支持updateById(id)与update(entity,wrapper)方法；

5、在update(entity,wrapper)方法下，wrapper不能复用。

|  |
| --- |
| @EnableTransactionManagement @Configuration // 在配置类中添加Mapper扫描，此时可以不用在主启动类中添加该注解 @MapperScan("com.guli.mp.mapper") public class MPConfig {  // 乐观锁插件  @Bean  public OptimisticLockerInterceptor optimisticLockerInterceptor(){  return new OptimisticLockerInterceptor();  }  } |

测试乐观锁，此时因为user和user2获取的数据相同，即其version相同，因此在执行updateById方法时，user更新成功，user2更新失败。

|  |
| --- |
| @Test public void updateUserDataById(){  User user = userMapper.selectById(1167251854343761922L);  User user2 = userMapper.selectById(1167251854343761922L);  user.setName("True");  userMapper.updateById(user);  user2.setName("False");  userMapper.updateById(user2); } |

### select查询操作

|  |
| --- |
| @Test public void selectUserBtIds(){  // selectById方法表示通过一个id查询数据  User user = userMapper.selectById(1167253002278584322L);  System.*err*.println(user);  ArrayList<Long> list = new ArrayList<>();  list.add(1167252359367335937L);  list.add(1167252359568662529L);  list.add(1167253002278584322L);  // selectBatchIds方法表示通过多个id查询数据，这些id可以  List<User> users = userMapper.selectBatchIds(list);  users.forEach(System.*err*::println);  List<User> users2 = userMapper.selectBatchIds(Arrays.*asList*(  1167253002278584322L,  1167252359367335937L,  1167252359568662529L));  users2.forEach(System.*err*::println);  Map<String,Object> map = new HashMap<>();  map.put("name", "Tom");  map.put("age", 0);  // 通过map封装查询条件进行查询，map中的key对应表中的字段名  // 能够将驼峰式命名的属性和下划线命名的字段进行互相转换  List<User> users3 = userMapper.selectByMap(map);  users3.forEach(System.*err*::println); } |

## MyBatis Plus功能

### 自动填充功能

在实际开发中，数据库中的某些数据经常需要以相同的方式填充，如创建时间、更新时间或删除时间等，此时就可以使用MyBatis Plus的自动填充功能来完成这些字段的赋值。

首先在user数据库表中添加新的字段create\_time和update\_time,然后在对应的JavaBean中添加属性createTime和updateTime，在这些属性上添加@TableField注解并设置填充时机。

|  |
| --- |
| @Data public class User {  @TableId(type = IdType.*ID\_WORKER*)  private Long id;  private String name;  private Integer age;  private String email;  // FieldFill.INSERT表示该属性在插入时被 MetaObjectHandler 拦截进行填充操作  @TableField(fill = FieldFill.*INSERT*)  private Date createTime;  // FieldFill.INSERT\_UPDATE表示该属性在插入或更新时被 MetaObjectHandler 拦截进行填充操作  @TableField(fill = FieldFill.*INSERT\_UPDATE*)  private Date updateTime; } |

创建自定义类并实现MyBatis Plus中的MetaObjectHandler接口，然后重写updateFill和insertFill方法，最后在该接口上添加@Component注解使其能够被Spring容器扫描到并添加到容器中。

|  |
| --- |
| @Component public class DateMetaObjectHandler implements MetaObjectHandler {  @Override  public void updateFill(MetaObject metaObject) {  this.setFieldValByName("updateTime", new Date(),metaObject);  }  @Override  public void insertFill(MetaObject metaObject) {  this.setFieldValByName("createTime", new Date(), metaObject);  this.setFieldValByName("updateTime", new Date(), metaObject);  } } |

配置完成后再执行insert或update操作时就会自动在该数据库表中的create\_time和update\_time字段中填充数据。

还有另一种自动填充方案，就是在创建数据库表时将create\_time和update\_time字段分别设置默认值来达到数据级别的自动填充，其SQL语句如下：

|  |
| --- |
| `create\_time` datetime DEFAULT *CURRENT\_TIMESTAMP* ON UPDATE *CURRENT\_TIMESTAMP*, `update\_time` datetime DEFAULT *CURRENT\_TIMESTAMP* ON UPDATE *CURRENT\_TIMESTAMP*, |

### 分页功能

MyBatis Plus自带分页插件，只要简单的配置就可实现分页功能。在配置类中使用@Bean注解标注在方法上，以此将分页插件注册到Spring容器中。

|  |
| --- |
| // 分页插件 @Bean public PaginationInterceptor paginationInterceptor(){  return new PaginationInterceptor(); } |

测试查询分页的效果

|  |
| --- |
| @Test public void selectUserByPage(){  // Page构造器的第一个参数表示当前页码，第二个参数表示每页显示的记录数  Page<User> page = new Page<>(1,3);  userMapper.selectPage(page, null);  // getRecords方法表示获取符合条件的记录数集合  List<User> users = page.getRecords();  users.forEach(System.*err*::println);  // 使用selectMapsPage分页时必须使用IPage的实现类的实例来调用getRecords方法  // 否则会发生数据类型转换异常  IPage<Map<String,Object>> mapIPage = userMapper.selectMapsPage(page, null);  mapIPage.getRecords().forEach(System.*err*::println);  System.*err*.println("总页数：" + page.getPages());  System.*err*.println("总记录数：" + page.getTotal());  System.*err*.println("当前页码：" + page.getCurrent());  System.*err*.println("是否有下一页：" + page.hasNext());  System.*err*.println("是否有上一页：" + page.hasPrevious()); } |

### 逻辑删除功能

物理删除：真实删除，将对应数据从数据库中删除，之后查询不到此条被删除数据。

逻辑删除：假删除，将对应数据中代表是否被删除字段状态修改为“被删除状态”，之后在数据库中仍旧能看到此条数据记录。

在企业中删除不是从数据库直接删除的物理删除，而是逻辑删除！即在企业中使用update操作修改is\_delete字段的值！

在数据库中添加is\_delete字段

|  |
| --- |
| alter table `mp`.`user` add column `is\_delete` tinyint(2) null; |

在JavaBean中添加isDelete属性，并使用@TableField和@TableLogic注解标注。

|  |
| --- |
| // FieldFill.INSERT表示该属性在插入时被 MetaObjectHandler 拦截进行初始化操作 @TableField(fill = FieldFill.*INSERT*) // 该注解表示逻辑删除，和配置文件中配置的删除和不删除状态进行绑定 @TableLogic private Boolean isDelete; |

在Spring Boot配置文件中添加相关配置，使JavaBean中的Boolean型属性isDelete能够与user表中的tinyint类型的is\_delete字段对应上。该配置为默认配置，如果默认值和MyBatis Plus相同则可以不用配置。

|  |
| --- |
| #指定删除和不删除的状态 mybatis-plus.global-config.db-config.logic-delete-value=1 mybatis-plus.global-config.db-config.logic-not-delete-value=0 |

在配置类中配置逻辑删除的相关插件

|  |
| --- |
| // 逻辑删除插件 @Bean public ISqlInjector sqlInjector(){  return new LogicSqlInjector(); } |

在配置完成后，MyBatis Plus中其他任何操作都会自动添加逻辑删除字段的判断，即在执行的SQL语句中的where后添加is\_delete=0。

### 性能分析功能

性能分析拦截器用于输出执行的每条SQL语句及其执行时间，SQL性能执行分析仅推荐在开发环境中使用，通过设置指定时间，一旦执行的SQL语句使用的时间超过指定时间，就会停止运行。该功能有助于发现问题。

在配置类中配置性能分析相关的插件

|  |
| --- |
| //SQL执行性能分析插件，用于开发和测试环境，不要用于线上环境 @Bean @Profile({"dev","test"}) // 设置作用的环境，此时作用于开发和测试环境 public PerformanceInterceptor performanceInterceptor(){  PerformanceInterceptor performanceInterceptor = new PerformanceInterceptor();  // 设置SQL最大执行时长，超过设置的时长则不执行，单位为毫秒  performanceInterceptor.setMaxTime(10000000);  // 设置SQL是否格式化，默认为false  performanceInterceptor.setFormat(true);  return performanceInterceptor; } |

在Spring Boot的配置文件中配置环境，可以针对不同的环境配置不同的Spring Boot配置文件application-dev.properties、application-test.properties、application-prod.properties，也可以自定义环境名称如test1、test2。

|  |
| --- |
| # 开发环境设置:dev||test||prod spring.profiles.active=dev |

配置完成后执行SQL语句，如果时间超过设置的时间，则会抛出异常：The SQL execution time is too large。

## MyBatis Plus复杂查询

如果想进行复杂条件查询，那么需要使用条件构造器 Wapper，涉及到如下方法：delete、selectOne、selectCount、selectList、selectMaps、selectObjs和update。

**Wrapper类结构**

Wrapper 条件构造抽象类，最顶端父类

|--AbstractWrapper 用于查询条件封装，生成 sql 的 where 条件

|--QueryWrapper Entity对象封装操作类，不是用lambda语法

|--UpdateWrapper Update条件封装，用于Entity对象更新操作

|--AbstractLambdaWrapper Lambda语法使用 Wrapper统一处理解析 lambda 获取 column

|--LambdaUpdateWrapper Lambda语法查询封装Wrapper

|--LambdaQueryWrapper Lambda语法更新封装Wrapper

注意：以下条件构造器的方法入参中的 column 均表示数据库字段

### ge、gt、le、lt、isNull、isNotNull

|  |
| --- |
| @Test public void testDelete() {  QueryWrapper<User> queryWrapper = new QueryWrapper<>();  queryWrapper  .isNull("name")  .ge("age", 12)  .isNotNull("email");  int result = userMapper.delete(queryWrapper);  System.*out*.println("delete return count = " + result); } |

SQL：UPDATE user SET deleted=1 WHERE deleted=0 AND name IS NULL AND age >= ? AND email IS NOT NULL

### eq、ne

注意：seletOne返回的是一条实体记录，当出现多条时会报错

|  |
| --- |
| @Test public void testSelectOne() {  QueryWrapper<User> queryWrapper = new QueryWrapper<>();  queryWrapper.eq("name", "Tom");  User user = userMapper.selectOne(queryWrapper);  System.*out*.println(user); } |

SELECT id,name,age,email,create\_time,update\_time,deleted,version FROM user WHERE deleted=0 AND name = ?

### between、notBetween

包含大小边界

|  |
| --- |
| @Test public void testSelectCount() {  QueryWrapper<User> queryWrapper = new QueryWrapper<>();  queryWrapper.between("age", 20, 30);  Integer count = userMapper.selectCount(queryWrapper);  System.*out*.println(count); } |

SELECT COUNT(1) FROM user WHERE deleted=0 AND age BETWEEN ? AND ?

### allEq

|  |
| --- |
| @Test public void testSelectList() {  QueryWrapper<User> queryWrapper = new QueryWrapper<>();  Map<String, Object> map = new HashMap<>();  map.put("id", 2);  map.put("name", "Jack");  map.put("age", 20);  queryWrapper.allEq(map);  List<User> users = userMapper.selectList(queryWrapper);  users.forEach(System.*out*::println); } |

SELECT id,name,age,email,create\_time,update\_time,deleted,version

FROM user WHERE deleted=0 AND name = ? AND id = ? AND age = ?

### like、notLike、likeLeft、likeRight

selectMaps返回Map集合列表

|  |
| --- |
| @Test public void testSelectMaps() {  QueryWrapper<User> queryWrapper = new QueryWrapper<>();  queryWrapper  .notLike("name", "e")  .likeRight("email", "t");  List<Map<String, Object>> maps = userMapper.selectMaps(queryWrapper);//返回值是Map列表  maps.forEach(System.*out*::println); } |

SELECT id,name,age,email,create\_time,update\_time,deleted,version

FROM user WHERE deleted=0 AND name NOT LIKE ? AND email LIKE ?

### in、notIn、inSql、notinSql、exists、notExists

in、notIn：

notIn("age",{1,2,3})--->age not in (1,2,3)

notIn("age", 1, 2, 3)--->age not in (1,2,3)

inSql、notinSql：可以实现子查询

例: inSql("age", "1,2,3,4,5,6")--->age in (1,2,3,4,5,6)

例: inSql("id", "select id from table where id < 3")--->id in (select id from table where id < 3)

|  |
| --- |
| @Test public void testSelectObjs() {  QueryWrapper<User> queryWrapper = new QueryWrapper<>();  //queryWrapper.in("id", 1, 2, 3);  queryWrapper.inSql("id", "select id from user where id < 3");  List<Object> objects = userMapper.selectObjs(queryWrapper);//返回值是Object列表  objects.forEach(System.*out*::println); } |

SELECT id,name,age,email,create\_time,update\_time,deleted,version

FROM user WHERE deleted=0 AND id IN (select id from user where id < 3)

### or、and

注意：这里使用的是UpdateWrapper，不调用or则默认为使用and连接。

|  |
| --- |
| @Test public void testUpdate1() {  //修改值  User user = new User();  user.setAge(99);  user.setName("Andy");  //修改条件  UpdateWrapper<User> userUpdateWrapper = new UpdateWrapper<>();  userUpdateWrapper  .like("name", "h")  .or()  .between("age", 20, 30);  int result = userMapper.update(user, userUpdateWrapper);  System.*out*.println(result); } |

UPDATE user SET name=?, age=?, update\_time=? WHERE deleted=0 AND name LIKE ? OR age BETWEEN ? AND ?

### 嵌套or、嵌套and

这里使用了lambda表达式，or中的表达式最后翻译成sql时会被加上圆括号

|  |
| --- |
| @Test public void testUpdate2() {  //修改值  User user = new User();  user.setAge(99);  user.setName("Andy");  //修改条件  UpdateWrapper<User> userUpdateWrapper = new UpdateWrapper<>();  userUpdateWrapper  .like("name", "h")  .or(i -> i.eq("name", "李白").ne("age", 20));  int result = userMapper.update(user, userUpdateWrapper);  System.*out*.println(result); } |

UPDATE user SET name=?, age=?, update\_time=? WHERE deleted=0 AND name LIKE ?

OR ( name = ? AND age <> ? )

### orderBy、orderByDesc、orderByAsc

|  |
| --- |
| @Test public void testSelectListOrderBy() {  QueryWrapper<User> queryWrapper = new QueryWrapper<>();  queryWrapper.orderByDesc("id");  List<User> users = userMapper.selectList(queryWrapper);  users.forEach(System.*out*::println); } |

SELECT id,name,age,email,create\_time,update\_time,deleted,version

FROM user WHERE deleted=0 ORDER BY id DESC

### last

直接拼接到 sql 的最后，注意只能调用一次,多次调用以最后一次为准 有sql注入的风险,请谨慎使用。

|  |
| --- |
| @Test public void testSelectListLast() {  QueryWrapper<User> queryWrapper = new QueryWrapper<>();  queryWrapper.last("limit 1");  List<User> users = userMapper.selectList(queryWrapper);  users.forEach(System.*out*::println); } |

SELECT id,name,age,email,create\_time,update\_time,deleted,version

FROM user WHERE deleted=0 limit 1

### 指定要查询的列

|  |
| --- |
| @Test public void testSelectListColumn() {  QueryWrapper<User> queryWrapper = new QueryWrapper<>();  queryWrapper.select("id", "name", "age");  List<User> users = userMapper.selectList(queryWrapper);  users.forEach(System.*out*::println); } |

SELECT id,name,age FROM user WHERE deleted=0

### set、setSql

最终的sql会合并 user.setAge()，以及 userUpdateWrapper.set()  和 setSql() 中 的字段

|  |
| --- |
| @Test public void testUpdateSet() {  //修改值  User user = new User();  user.setAge(99);  //修改条件  UpdateWrapper<User> userUpdateWrapper = new UpdateWrapper<>();  userUpdateWrapper  .like("name", "h")  .set("name", "老李头")//除了可以查询还可以使用set设置修改的字段  .setSql(" email = '123@qq.com'");//可以有子查询  int result = userMapper.update(user, userUpdateWrapper); } |

UPDATE user SET age=?, update\_time=?, name=?, email = '123@qq.com' WHERE deleted=0 AND name LIKE ?